Suoritusanomalioiden havaitseminen lokitiedostoista ohjaamattoman koneoppimisen menetelmin

Anniina Sallinen

Helsinki 19.5.2018

Pro gradu -tutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen osasto

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Studieprogram — Study Programme
Matemaattis-luonnontieteellinen	Tietojenkäsittelytiede

Tekijä — Författare — Author
Anniina Sallinen

Työn nimi — Arbetets titel — Title

Suoritusanomalioiden havaitseminen lokitiedostoista ohjaamattoman koneoppimisen menetelmin

Ohjaajat — Handledare — Supervisors
Jyrki Kivinen, Tommi Mikkonen ja Teppo Ahonen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Pro gradu -tutkielma	19.5.2018	56 sivua

Tiivistelmä — Referat — Abstract

Tutkielmassa esitellään menetelmiä, joilla lokitiedostoista voidaan havaita suoritusanomalioita. Kyseessä on monivaiheinen prosessi, ja ensimmäisenä suoritetaan lokipohjien jäsentäminen joko lähdekoodista tai lokitiedostoista. Lähdekoodista jäsentäminen voidaan suorittaa melko suoraviivaisesti, koska monissa ohjelmointikielissä lokiviestipohjat ovat erotettavissa suoraan tulostuslausekkeista. Lokitiedostosta jäsentäminen ei ole yhtä suoraviivaista, ja siihen käytetäänkin usein ryvästämistä. Ryvästämistä käyttävät esiteltävät menetelmät ovat nimeltään SLCT ja LKE. Jäsentämisen tarkoituksena on erottaa lokiviesteistä vakio-osa ja muuttujat, koska sama vakio lokiviesteissä kertoo siitä, että sama tulostuslauseke on tuottanut viestit. Vakioiden perusteella luodaan säännölliset lausekkeet, ja niillä korvataan lokitiedostojen riveillä olevat lokiviestit.

Rivien korvaamisen jälkeen tiedostot voidaan jakaa sarjoihin aika- tai tunnisteperusteisesti. Aikaperusteisesti jaetusta datasta voidaan tutkia sitä, toimiiko järjestelmä yhdenmukaisesti eri ajanjaksojen välillä, ja tunnisteperusteisesti jaettaessa pyritään muodostamaan suorituspolkuja tunnisteiden perusteella. Lokijaksot muutetaan vielä numeeriseen muotoon esimerkiksi laskemalla erilaisten tilojen esiintymistä ajanjakson sisällä tai laskemalla eri viestityyppien määrä ryhmässä, joka sisältää viestejä, joissa esiintyy tietty tunniste.

Esiprosessoinnin lopputulos toimii syötteenä koneoppimismenetelmille. Ohjaamattoman koneoppimisen menetelmistä esitellään ryvästäminen, painotusmenetelmät, pääkomponenttianalyysi ja invariantit. Ryvästäminen jakaa datapisteet ryväksiin tietyn kriteerin perusteella. Menetelmää varten tulee määritellä, miten kahden datapisteen ja kahden rypään välinen etäisyys määritellään. Kahden datapisteen välinen etäisyys voi olla esimerkiksi kosinietäisyys, ja kahden rypään välinen etäisyys voidaan määritellä esimerkiksi niiden kauimmaisten datapisteiden välisenä etäisyytenä. Painotusmenetelmissä pyritään korostamaan datapisteen jotakin ominaisuutta, kuten entropiaa. Pääkomponenttianalyysi pyrkii taas löytämään datasta merkittävimmät piirteet, ja sen perusteella tunnistamaan myös poikkeavuudet. Invarianttimenetelmä kuvaa järjestelmän suorituspolkuja erilaisina yhtälöinä ja kerroinvektoreina. Lokeista pyritään louhimaan harvoja ja kompakteja kerroinvektoreita, jotka sisältävät vain kokonaislukuja.

Menetelmiä vertailtiin yleistettävyyden, suorituskyvyn ja suoriutumisen perusteella. Näiden kriteerien perusteella lupaavimmaksi yhdistelmäksi muodostui SLCT lokiviestien esiprosessointiin ja invarianttimenetelmä koneoppimismenetelmäksi. SLCT on yleistettävämäpi kuin lähdekoodista jäsentäminen, ja tarkempi sekä tehokkaampi kuin LKE. Invarianttimenetelmä suoriutui vertailevassa tutkimuksessa hyvin eri järjestelmissä, joten se on yleistettävä ja tarkka menetelmä. Se ei ole tehokkain esitelty koneoppimismenetelmä, mutta sen suoritusaikaa voidaan parantaa erilaisilla optimoinneilla.

ACM Computing Classification System (CCS):
Information systems → Information systems applications → Data mining
Computing methodologies → Machine learning → Unsupervised Learning → Anomaly Detection

Avainsanat — Nyckelord — Keywords
anomalioiden havaitseminen, ohjaamaton koneoppiminen, lokitiedostot

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto

Järjestelmissä sattuvat suoritusaikaiset virheet voivat tulla kalliiksi yrityksille. Virheen sattuessa sen syytä lähdetään yleensä jäljittämään lokitiedostoista erilaisten avainsanojen ja hakutoimintojen avulla. Avainsana voi olla esimerkiksi 'ERROR' tai 'EXCEPTION', mutta lokitiedoston läpikäyminen avainsanojen avulla on tehotonta eikä välttämättä paljasta virheen syytä. Lisäksi voidaan määrittää lista hienostuneempia avainsanoja, joita etsiä, mutta tällaisten listojen ylläpito on työlästä. Lokitiedostot tuottanut järjestelmä voi olla esimerkiksi hajautettu varastojärjestelmä, kuten HDFS, tai data-alusta, joka kerää dataa eri lähteistä, prosessoi sitä ja tallentaa sen tietovarastoon myöhempää käyttöä varten. Kyseessä voi oikeastaan olla mikä tahansa järjestelmä, joka tuottaa suuret määrät lokidataa. Tuotetalo voi soveltaa tuotteelle räätälöityjä ratkaisuja, mutta konsulttiyrityksen kannalta olisi hyvä, jos sama menetelmä toimisi eri järjestelmissä. Näin samalla työkalulla voitaisiin tunnistaa anomalioita riippumatta järjestelmästä. Konsulttiyritykselle tärkeitä menetelmän ominaisuuksia on siis se, että se ei tee liikaa oletuksia lokiviestien formaatista, jotta se voisi toimia eri järjestelmissä, ja että menetelmä on tehokas, jotta anomalioiden havaitsemisessa ei olisi pitkää viivettä ja virheet päästäisiin korjaamaan pian. Tutkielmassa käytetään siis näitä kriteerejä menetelmien arvioimiseen.

Järjestelmän lokitiedostot ovat paras paikka lähteä jäljittämään virheen syytä, sillä järjestelmän kehittäjät lisäävät lokituslausekkeita tärkeäksi näkemiinsä kohtiin lähdekoodissa. Lokitiedostojen ja niistä saatavan informaation tehokkaan hyödyntämisen merkitys korostuu, mitä suuremmista järjestelmistä on kyse. Lokitiedostojen manuaalinen läpikäyminen on työlästä ja aikaavievää, ja vaikka käytössä olisi hälytysjärjestelmä, ei tapahtuman syytä ole välttämättä helppo jäljittää. Syy virheeseen voi olla vaikeasti havaittava asia, kuten suorituskykyongelmat. Mitä suuremmaksi järjestelmä kasvaa, sitä enemmän se tuottaa lokeja, ja jos kyseessä on ryväs, sen jokainen solmu tuottaa rinnakkain lokeja. Virheet esiintyvät lokitiedostoissa usein anomalioina eli odotetusta hahmosta poikkeavana datapisteenä tai sarjana pisteitä. Manuaalisesti anomalioita voi olla hankalaa havaita osittain valtavien datamassojen vuoksi ja osittain myös siksi, että järjestelmä pitäisi tuntea hyvin huomatakseen poikkeavuudet. Anomalioiden havaitsemismenetelmillä voidaan havaita virheitä, jotka eivät vielä haittaa järjestelmän toimintaa näkyvästi, ja korjata virheet ennen kuin ne aiheuttavat ongelmia.

Tämän tutkielman tarkoituksena on vertailla poikkeavuuksien havaitsemiseen soveltuvia, mahdollisimman automatisoituja menetelmiä. Järjestelmien generoimat lokiviestit eivät sisällä tietoa siitä, onko kyseessä poikkeavuus vai ei, joten ohjaamaton oppiminen on luontevin koneoppimisen haara tähän tarkoitukseen. Jos haluttaisiin käyttää ohjatun koneoppimisen menetelmiä, pitäisi alkuun tehdä valtava työ lokiviestien manuaalisesti luokittelemiseksi. Luokittelua voitaisiin tehdä esimerkiksi määrittelyjä vasten tai luomalla lista tapahtumista, jotka voivat viitata virheeseen. Nämä tavat eivät kuitenkaan ole usein mahdollisia, koska määrittelyt saattavat olla puutteellisia tai epäjohdonmukaisia ja koska manuaalinen virhetilanteiden määrittely ja ylläpito on työlästä (Mariani & Pastore, 2008). Ohjaamattoman koneoppimi-

sen menetelmillä tehdyt ratkaisut ovat myös yleistettävissä erilaisiin järjestelmiin, toisin kuin ratkaisut, jotka nojaavat sovelluskohtaiseen tietoon. Ohjaamattoman koneoppimisen menetelmät, joilla pyritään havaitsemaan anomaliat, noudattavat usein samaa prosessia. Ensin lokit kerätään tiedostoon, josta ne jäsennetään (parse) lokiviestien malleiksi. Tässä voidaan myös tehdä muuta esiprosessointia. Seuraavaksi tehdään piirteiden valinta ja muutetaan data koneoppimismenetelmille sopivampaan muotoon. Koneoppimisen tuottamien mallien avulla suoritetaan sitten itse anomalioiden havaitseminen. Tämä prosessi on näkyvissä kuvassa 1.

Tutkielma on kirjallisuuskatsaus tutkimuksista, joissa on käytetty ohjaamattoman koneoppimisen menetelmiä anomalioiden tunnistamiseen lokitiedostoista. Luku 2 käsittelee koneoppimisen ja anomalioiden havaitsemisen pohjateoriaa. Lokien tuottamista ja niiden merkitystä järjestelmissä käsittelee luku 3. Samassa esitellään myös datan esiprosessointia, kun kyseessä on lokiviestit: mistä lokiviestipohjia tuotetaan ja mitä etuja tai haasteita erilaisiin tapoihin liittyy. Ongelma-alueeseen sovellettuja keskeisiä koneoppimismenetelmiä esitellään luvussa 4. Luvussa 5 käydään läpi tapauksia, joissa esiteltyjen koneoppimismenetelmien tarkkuutta on tutkittu. Samassa luvussa käydään myös läpi eri menetelmien etuja ja haasteita laajemmin. Luvussa pohditaan, millaisiin sovelluksiin erilaiset menetelmät ovat sovellettavissa ja onko jokin menetelmä soveltuvampi laajemmalle skaalalle sovelluksia kuin toiset. Luku 7 on yhteenveto, eli siinä kootaan käsitelty asia ja esitellään, mitkä menetelmät ovat lupaavimpia.

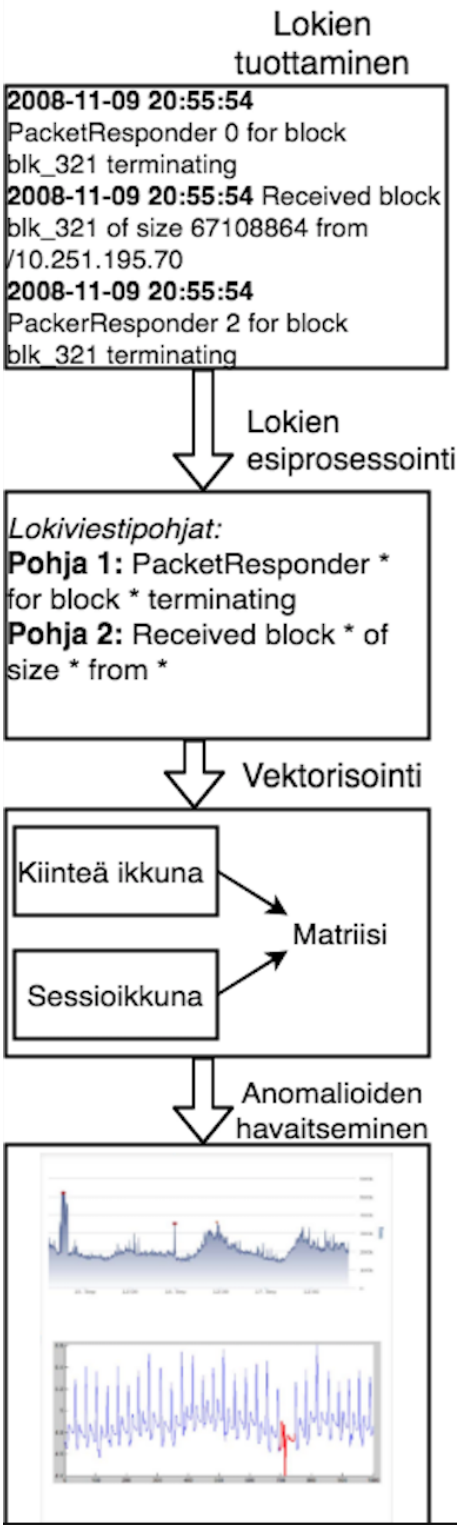
2 Taustateoria

Tutkielman keskeisiä aihepiirejä ovat koneoppiminen ja anomalioiden havaitseminen. Erityisen keskeistä on ymmärtää, miten ohjaamaton ja ohjattu koneoppiminen eroavat toisistaan ja miten anomaliat datassa ovat sidoksissa sovellusalueeseen.

2.1 Koneoppiminen

Koneoppiminen määritellään kokoelmana menetelmiä, jotka voivat automaattisesti tunnistaa kaavoja datassa ja tehdä sen perusteella ennusteita tulevaisuuteen tai suorittaa muun tyyppistä päätöksentekoa huomioiden epävarmuuden. Koneoppimisessa syötteenä on joukko havaintoja $X = \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_m\}$, jossa $\bar{x}_i \in \mathbb{R}^n$ ja n on piirteiden määrä. Koneoppiminen jaetaan kolmeen osa-alueeseen: ohjattuun ja ohjaamattomaan koneoppimiseen, sekä palauteoppimiseen (Robert, 2014). Palauteoppimista ei käsitellä tässä tutkielmassa tarkemmin.

Ohjatun koneoppimisen tapauksessa on havaintodataa, joka koostuu (syöte, tulos)-pareista. Olemassa olevan datan perusteella opitaan syötteen ja tuloksen välinen suhde ja muodostetaan malli. Datan voi myös jakaa opetus- ja testausdataan ja käyttää testausdatan mallin validoimiseen. Formaalisti ohjatussa koneoppimisessa käytettävä datajoukko (dataset) on $\mathcal{D} = \{(\boldsymbol{x}_i, y_i) \mid i = 1 \ldots m\}$, jossa x_i on syöte-



Kuva 1: Anomalioiden havaitsemisen prosessi (S. He ja muut, 2016).

data eli selittävä muuttuja, y_i on tulos eli vastemuuttuja ja m on datarivien määrä. Ohjatun koneoppimisen tehtävänä on minimoida globaali tappiofunktio (global loss function) säätämällä mallin parametreja ja samalla välttää ylisovittaminen, jotta malli toimii ennen näkemättömilläkin datapisteillä. Ohjatun koneoppimisen tuloksena syntyvä malli voi tuottaa jatkuvia numeerisia arvoja kuten luku väliltä 0 ja 1 tai kategorioita kuten jokin väri. Jatkuvien numeeristen arvojen tapauksessa kyseessä on regressio ja kategorioiden tapauksessa luokittelu (Bonaccorso, 2017).

Ohjaamattoman koneoppimisen tapauksessa syötteenä on joukko havaintoja $D = \{x_i \mid i = 1 \dots m\}$, mutta ei tulosta y_i. Esimerkiksi regressiota ei voida sovittaa, koska ennustettavaa vastemuuttujaa ei ole (Tibshirani, James, Witten, & Hastie, 2013). Ohjaamattoman koneoppimisen tehtävänä on löytää mielenkiintoisia kaavoja datasta. Kaavoja tai niiden tyyppiä ei ole etukäteen määritelty, eikä samaa virhemetriikkaa voida soveltaa kun ohjatun koneoppimisen tapauksessa. Ohjaamattoman koneoppimisen avulla voidaan esimerkiksi tunnistaa rypäitä, piileviä tekijöitä (latent factors) tai verkkojen rakenteita (Bonaccorso, 2017).

2.2 Anomalioiden havaitseminen

Hahmontunnistus (pattern recognition) on koneoppimisen haara, jossa keskitytään kaavojen ja säännöllisyyksien tunnistamiseen datasta. Sitä käytetään sekä ohjatussa että ohjaamattomassa koneoppimisessa, ja sen muodostama malli voi olla esimerkiksi piilotettu Markovin malli, ja esimerkkimenetelmä ryvästäminen. Hahmontunnistusta voidaan suorittaa yksittäiselle datapisteelle tai sarjalle datapisteitä. Tässä kirjallisuudessa käsitellyt menetelmät perustuvat hahmojen ja niistä poikkeavien datapisteiden tunnistamiseen.

Tunnistetut poikkeavuudet ovat anomalioita. Anomalialla tarkoitetaan odotetusta hahmosta poikkeavaa datapistettä tai sarjaa pisteitä (Chandola, Banerjee, & Kumar, 2009). Anomalian lisäksi voidaan käyttää termiä poikkeavuus (outlier), minkä lisäksi kohina (noise) on hyvin lähellä oleva käsite. Kohinan ja anomalian ero on siinä, että anomalia nähdään kiinnostavana asiana, joka tarjoaa informaatiota. Kohina taas usein on epäkiinnostavaa ja se halutaan poistaa datan joukosta.

Anomalioiden havaitsemisen sovelluksia ovat muun muassa järjestelmään tunkeutumisen havaitseminen ja pankkikorttihuijausten havaitseminen (Chandola ja muut, 2009). Lisäksi ongelmaan kehitettyjä menetelmiä voidaan soveltaa myös järjestelmävirheiden tunnistamiseen. Useimmat anomalioiden havaitsemismenetelmät ovat sovelluskeskeisiä ja soveltuvat tietyn ongelman ratkaisuun. Erityisesti datan ja tarkasteltavien anomalioiden tyyppi vaikuttaa siihen, millaisia menetelmiä voidaan käyttää. Anomalian tyyppi voi olla esimerkiksi pisteanomalia tai yhteydestä riippuva anomalia, jolloin jonkin tapahtuman poikkeuksellisuus liittyy kontekstiin. Esimerkiksi ajanjaksodatassa aikaikkuna on kontekstuaalinen ominaisuus, joka määrittää tapahtuman poikkeavuuden.

Anomalioiden tunnistamista voidaan tehdä ohjatun, ohjaamattoman ja puoliohjatun koneoppimisen keinoin (Chandola ja muut, 2009). Kuten aiemmin mainit-

tiin, ohjatun koneoppimisen menetelmät olettavat, että käytettävissä on harjoitusdataa, joka on jo valmiiksi luokiteltu. Yleensä tällöin muodostetaan ennustava malli, joka pyrkii erottamaan anomaliat normaalista datasta. Ohjaamattoman koneoppimisen menetelmissä oletetaan, että anomalioita on vain vähän suhteessa normaaliin dataan. Jos tämä oletus ei pidä paikkaansa, tuloksena on paljon väärä positiivinen -tapauksia. Näiden kahden haaran lisäksi on puoliohjattu oppiminen. Puoliohjatun oppimisen tapauksessa harjoitusdatassa on luokittelu vain normaaleille tapahtumille. Tämän perusteella voidaan luoda malli, joka tunnistaa normaalit tapahtumat ja pystyy siten erottelemaan anomaliat. Monia puoliohjatun oppimisen menetelmiä voidaan mukauttaa toimimaan ohjaamattoman oppimisen tavoin. Tällöin käytetään luokittelematonta dataa ja oletetaan, että se sisältää hyvin vähän anomalioita ja että malli pystyy käsittelemään tämän asian.

Tavoitteena on sekä löytää kaikki anomaliat että minimioida väärien hälytysten määrä, mutta yleensä käytetyt menetelmät joutuvat tekemään kompromisseja. Jos kaikki anomaliat halutaan löytää, esiintyy yleensä myös vääriä positiivisia enemmän, ja jos taas väärien positiivisten määrä halutaan minimoida, osa anomalioista saattaa jäädä huomaamatta (Chandola ja muut, 2009).

2.3 Lokiviestit

Lokit tallentavat tietoa järjestelmien suoritusaikaisista tapahtumista. Niiden käyttö on vakiintunutta, ja ohjelmoijat lisäävät lokituslausekkeita tärkeäksi näkemiinsä kohtiin koodissa. Lokidata on osittain strukturoimatonta tekstidataa luonnollisella kielellä. Tulostuslausekkeet lähdekoodissa näyttävät tyypillisesti seuraavanlaiselta:

```
log(lokitaso, "logging message %s", muuttuja)
```

Lokitasolla ilmaistaan lokiviestin luonne, ja sen arvona voi olla esimerkiksi 'FATAL', 'ERROR' tai 'INFO'. Lokiviesti sisältää vakio-osan ja mahdollisesti myös muuttujan (Fu ja muut, 2014). Lokitiedostossa lokituslausekkeen lopputulos voi näyttää esimerkiksi seuraavalta:

2017-11-09 20:35:32,146 INFO Deployment successful, starting database on host 10.251.31.5

Yleensä siis lokiviestiin sisältyy myös aikaleima, jonka lokiviestin lisäämisen hoitava funktio lisää (P. He, Zhu, He, Li, & Lyu, 2016).

Lokeja käytetään järjestelmien hallinnassa laajasti, ja niiden avulla voidaan tehdä esimerkiksi anomalioiden tunnistamista, virheen etsimistä, suorituskykydiagnosointia ja työkuorman mallinnusta. Siksi onkin tärkeää olla lokittamatta liikaa tai liian vähän. Jos lokitusta lisätään liian useaan kohtaan järjestelmässä, se alkaa vaikuttamaan suoritustehoon, koska se lisää suorittimen käyttöä ja I/O -operaatioita. Lisäksi saatetaan tuottaa paljon hyödytöntä tai tarpeetonta informaatiota, jos lokituslausekkeita lisätään joka puolelle järjestelmää. Tarpeeton lokitus aiheuttaa sen, että

tärkeän informaation löytäminen on hankalaa. Lokien lisääminen ja ylläpito lisäksi vaativat kehittäjien resursseja (P. He ja muut, 2016). Jos lokituslausekkeita taas on liian vähän, virheen etsiminen vaikeutuu, kun kriittisistä virheistä ei ole lokiviestejä eikä suorituspolku ole pääteltävissä lokitiedostosta.

Lokitiedostot ovat osin strukturoimatonta dataa, koska lokiviestien vakio-osa on vapaamuotoinen merkkijono eikä muuttujien paikkaa lopullisessa lokiviestissä ole määrätty. Verrattuna esimerkiksi XML- tai JSON-muotoiseen dataan siitä on siis hankalaa erottaa eri osia. Lokiviestien esiprosessoinnille on esitetty useita tapoja, mutta useat tutkijat (Xu, Huang, Fox, Patterson, & Jordan, 2009a; Fu, Lou, Wang, & Li, 2009) ovat päätyneet eristämään siitä jo aiemmin mainitut muuttujat ja vakiot. Esimerkiksi lokiviestissä `'starting: database software_db in machine 123 is INITIALIZING.'` muuttujia ovat tietokannan nimi, palvelimen nimi ja nykyinen tila. Loppuosa viestistä on vakio. Muuttujat ovat erilaisia muuttuvia arvoja, jotka vaihtelevat eri solmujen (node) ja järjestelmän tilojen mukaan, mutta vakio on pysyvä eri solmujen ja järjestelmän eri tilojen tapauksissa.

Artikkelissa (Xu ja muut, 2009a) erotellaan muuttujista vielä kaksi tyyppiä: tunnisteet ja tilamuuttujat. Tunnisteen avulla tunnistetaan, mitä kohdetta (object) käsitellään, ja tilamuuttujat ovat luokkia, jotka kuvaavat kaikki mahdolliset tilat, joita kohteella voi olla. Tilamuuttujia voivat olla esimerkiksi "STARTING", "SAVING" ja "CLOSING", kun kohteena on tiedosto. Tilamuuttujilla on tyypillisesti vähemmän mahdollisia arvoja kuin tunnisteilla, joten ne voidaan erottaa tämän perusteella toisistaan.

Kun vakiot ja muuttujat on pystytty erottamaan toisistaan, luodaan niiden perusteella tyypillisesti lokiviestipohjat. Pohjat ovat säännöllisiä lausekkeita, joissa on näkyvissä vakio ja muuttujat on korvattu merkillä *. Näin pohjat saadaan riippumattomaksi solmusta ja järjestelmän sisäisestä tilasta ja lokiviestejä on helpompi vertailla (Fu ja muut, 2009).

3 Lokien tuottaminen ja esiprosessointi

Kun lokiviesteistä on saatu jäsennettyä pohjat, lokit tulee vielä muuttaa numeeriseen muotoon, jotta luvussa 4 esiteltävät koneoppimisalgoritmit pystyvät hyödyntämään niitä. Tätä varten ne täytyy jakaa tietyin kriteerein sarjoihin, esimerkiksi tunnisteen tai aikaleiman perusteella. Näiden sarjojen perusteella muodostetaan numeeriset piirrevektorit ja edelleen vielä matriisit. Vektorisoinnin syötteenä on siis lokiviestipohjat ja tuloksena piirrevektoreista muodostettu matriisi (S. He ja muut, 2016).

Tässä tutkielmassa käytettävässä materiaalissa anomalioiden tunnistus suoritetaan uusista lokiviesteistä ja se pohjautuu lokiviestipohjiin. Lokiviestejä ja niiden pohjia voidaan tuottaa eri tavoin eri lähteistä. Joissakin tapauksissa on käytetty väliohjelmistoja tuottamaan ennaltamäärätyn muotoisia lokiviestejä, joissakin menetelmissä

käytetään sovelluksen lähdekoodia ja joissakin louhitaan suoraan olemassa olevia lokitiedostoja. Seuraavissa osiossa esitellään näitä menetelmiä.

3.1 Lokilausekkeet lähdekoodissa

Tutkimuksen mukaan 96% kehittäjistä pitää lokituslausekkeita tärkeinä järjestelmän kehityksen ja ylläpidon kannalta (P. He ja muut, 2016). Tutkimuksessa tarkasteltiin sitä, mihin osiin järjestelmissä yleensä lisätään lokituslausekkeita ja miten päätöksentekoa lokituksesta voitaisiin automatisoida. Lokituksen päätöksenteon automatisointi ei kuulu tämän tutkielman piiriin, mutta anomalioiden tunnistamisen kannalta on tärkeää ymmärtää millaisia asioita lokitiedostoista yleensä löytyy. Tutkimus suoritettiin analysoimalla kahden Microsoftin järjestelmän lähdekoodeja ja haastattelemalla niiden kehittäjiä. Artikkelissa kutsutaan järjestelmiä Järjestelmä A:ksi ja Järjestelmä B:ksi, ja järjestelmä A:ssa oli 23 500 lokituslauseketta ja järjestelmässä B 95 300.

Satunnaisotannalla saadut 100 lokituslauseketta tutkittiin manuaalisesti ja jaettiin viiteen kategoriaan. Kategoriat ovat nähtävissä koodiesimerkissä 1. Ensimmäinen on väitteen tarkastuslokitus (assertion-checking logging). Tämä tarkoittaa sitä, että koodissa tarkastetaan jonkin väitteen paikkaansapitävyys, ja jos väite on epätosi, lokitetaan virheviesti automaattisesti. Tämän tyyppisiä lokituslausekkeita oli 19 sadasta. Toinen lokituslauseketyyppi on palautusarvotarkastuksen lokitus. Tähän kategoriaan kuuluvat lokituslausekkeet tarkistavat palautusarvon odottamattoman arvon, kuten nullin tai tyhjän merkkijonon tai listan, varalta. Odottamaton palautusarvo voi kertoa virheestä järjestelmässä. Sadasta tulostuslausekkeesta 14 oli palautusarvotarkastustyyppisiä. Poikkeuslokitus oli 27:llä esiintymisellä yleisin kategoria otannan lokituslausekkeista. Poikkeukset ovat useissa ohjelmointikielissä kuten Javassa ja C#:ssa sisäänrakennettuna ja lokituslauseke lisätään koodiin yleensä juuri ennen poikkeuksen heittämistä tai *catch*-lohkoon (P. He ja muut, 2016).

Kaikki edellä mainitut lokituslausekkeet liittyvät odottamattomiin tilanteisiin järjestelmässä, mutta lisäksi on kaksi kategoriaa, jonka lokituslausekkeet liittyvät suorituspisteisiin järjestelmässä. Ensimmäinen näistä on looginen haarautumislokitus ja toinen havannointipistelokitus. Näistä jälkimmäiseen kategoriaan kuuluvia lokituslausekkeita oli 24 sadasta ja ensimmäistä 16 sadasta. Haarautumislokitus liittyy koodissa esiintyviin *if*- ja *switch*-lausekkeisiin, ja auttaa suoritusaikaisen suorituspolun esittämisessä lokitiedostoissa. Myös havannointipistelokitus esittää järjestelmän suorituspolkua. Se saattaa lokittaa esimerkiksi funktion sisäänmeno- tai ulostulopisteen tai raportoida kriittisiä tapahtumia suorituksessa (P. He ja muut, 2016).

Tutkijat P. He ja muut tunnistivat vielä automaattisesti järjestelmien lähdekoodista eri lokituslausekkeiden kategoriat. Kategorian tunnistus perustui syntaksiin ja lokituslausekkeen rakenteeseen. Tarkempi kuvaus kriteereistä näkyy taulukossa 1. Järjestelmissä odottamattomia ja suorituspisteisiin liittyviä tilanteita lokittavia lokituslausekkeita oli lähes sama osuus. Järjestelmässä A odottamattomiin tilanteisiin liittyviä lokilausekkeita oli 12 525 (53%) ja järjestelmässä B 37 455 (39%).

```
// Väitteen tarkistuslokitus
assert(age < 0, "Ikä ei voi olla pienempi kuin nolla");

// Palautusarvon tarkastuslokitus
if(returnValue == null) {
        logger.info("Funktio X palautti arvon null");
}

// Poikkeuslokitus
try {
        bufferedReader=new BufferedReader(new File(fileName));
} catch(FileNotFoundException exception) {
        logger.warning("Haluttua tiedostoa {} ei löytynyt",
        ↪ fileName);
}

// Looginen haarautumislokitus
if(list.contains(element)) {
        log.trace("Lista sisälsi jo elementin {}, ei tarvetta lisätä
        ↪ uudelleen", element)
} else {
        log.trace("Lisätään uusi arvo {} listaan", element);
        list.add(element);
}

// Havaintopistelokitus
log.info("Lähetettiin henkilön {} tiedot palvelimelle {}", name,
↪ serverURL);
```

Koodiesimerkki 1: Esimerkit lokituskategorioista (P. He ja muut, 2016).

3.2 Lokitiedostoja tuottavat ulkoiset ohjelmistot

Sen lisäksi, että kehittäjät lisäävät lokituslausekkeita lähdekoodiin, voidaan lokitiedostoja tuottaa myös ulkoisilla ohjelmistoilla. Eräässä menetelmässä on käytetty toimijaa (agent), joka injektoidaan koodiin ja joka lokittaa jokaisen funktion suorituksen aloituksen ja päättymisen, ja aikaleiman näille tapahtumille (Mirgorodskiy, Maruyama, & Miller, 2006). Jos sovelluspalvelin vikaantuu tai tapahtuu jokin muu suoritusvirhe, lokitieto tallennetaan levylle, mistä luettuna se voidaan analysoida. Tarkemmin kuvattuna toimija injektoidaan käyttämällä kaappausmenetelmää (hijack method), ja se muuttaa sovelluksen binäärikoodia lisätäkseen lokituslausekkeita. Toimija injektoidaan jokaiseen prosessiin, joka on sovelluksessa käynnissä, niin että jokaisen niiden lokitus tapahtuu riippumattomasti toisista prosesseista. Näitä prosesseja valvoo koordinaattoriprosessi, joka tarkkailee deaktivointitapahtuman

Kategoria	**Kuvaus**
Tarkastuslokitus	Lokituksen suorittaa Assert-lauseke.
Palautusarvolokitus	Lokituslauseke on haarautumislausekkeen (*if, if-else, switch*) sisällä ja sisältää yhden tai useamman palautusarvon tarkistuksen. Lokituslauseke ei ole myöskään *catch*-lohkon sisällä.
Poikkeuslokitus	Lokituslauseke on *catch*-lohkon sisällä tai juuri ennen *throw*-lauseketta.
Looginen haarautumislokitus	Lokituslauseke on haarautumislausekkeen (*if, if-else, switch*) sisällä, ja lohkossa ei suoriteta paluuarvon tarkastamista.
Havaintopistelokitus	Kaikki loput lokituslausekkeet.

Taulukko 1: Lokituslausekkeiden automaattisen kategorisoinnin perusteet (P. He ja muut, 2016).

esiintymistä. Kun deaktivointitapahtuma esiintyy, lokitiedot tallennetaan levylle. Deaktivointitapahtuma voi olla esimerkiksi solmun kaatuminen tai muu käyttäjän deaktivointitapahtumaksi määrittelemä asia.

Pinpoint on binäärikoodiin injektoitava sovelluskehys, joka seuraa verkkoliikennettä ja pyrkii tunnistamaan siitä anomalioita (Chen, Kiciman, Fratkin, Fox, & Brewer, 2002). Sovelluskehys koostuu kolmesta alijärjestelmästä: monitoroitavan sovelluksen mukaan liitettävästä järjestelmästä, joka jäljittää sovellusten pyyntöjä ja vastauksia asettamalla kullekin pyynnölle tunnisteen; ulkopuolisesta järjestelmästä, joka seuraa verkkoliikennettä ja suorittaa vianmäärityksen, ja lisäksi tilastollisesta analysoijasta, joka pyrkii löytämään virheen aiheuttajat ryvästämisen avulla.

Sovellukseen liitettävä osa lisää tunnisteeseen tiedon järjestelmäkomponentista, joihin pyyntö on lähetetty, ja näin muodostaa listan kaikista komponenteista, joihin tietty pyyntö on kulkenut. Pinpointin prototyyppi, jota artikkeli käsittelee, toimii vain J2EE-sovelluksilla (nykyään Java EE), sillä pyyntöjen seuraaminen on toteutettu muokkaamalla J2EE-alustan olemassa olevaa toteutusta. Pyynnön tunniste talletetaan säiekohtaiseen muuttujaan ja lisätään HTTP-otsikkoon, jotta se kulkeutuu ulkopuoliselle virheenhavaitsijakomponentille. Lisäksi Pinpoint lokittaa virheet, jotka kulkeutuvat eri komponentteihin ja jotka ohjelmisto jättää näyttämättä käyttäjälle. Verkkoliikennettä seuraava osa on nimeltään Snifflet, ja se kaappaa TCP-paketteja. Se pyrkii havaitsemaan TCP- ja HTTP-virheet, jotka ilmenevät esimerkiksi pyyntöjen aikakatkaisuina tai HTTP 500 -statuksena. Pyynnöt lokitetaan joko onnistuneiksi tai epäonnistuneiksi. Nämä kaksi alijärjestelmää tuottavat määrätyn muotoisia lokitiedostoja, joissa on mukana pyynnön tunniste, tieto pyynnön onnistumisesta tai epäonnistumisesta ja missä komponenteissa pyyntö on käynyt.

3.3 Lokiviestipohjien eriyttäminen

Samojen lokitulostuskomentojen tuloksena syntyy usein samankaltaisia lokiviestejä, joten niitä vertailemalla voidaan erottaa saman tulostuslausekkeen tuottamien lokiviestien yhteinen osa, eli vakio. Vakion hyödyntäminen perustuu siihen, että eri lokituslausekkeet tuottavat erilaisia lokiviestejä, joten vakio-osan perusteella voidaan tunnistaa suorituspolkuja. Uniikkien vakioiden määrä on myös rajallinen ja pienempi kuin uniikkien lokiviestien määrä. Tämä pienentää ulottuvuuksien määrää myöhemmin (Fu ja muut, 2009).

Vakioiden tunnistamiseksi lokiviestit voidaan ryvästää (Bonaccorso, 2017). Ryvästystä varten LKE-nimisessä menetelmässä (Fu ja muut, 2009) on poistettu kaikista ilmeisimmät muuttujat empiiristen sääntöjen perusteella, koska paljon samoja muuttujia sisältävät lokiviestit saattavat olla samankaltaisia, vaikka olisivatkin eri tulostuslausekkeiden tuloksia. Lokiviesteissä tyypillisiä muuttujia ovat muun muassa URIt, IP-osoitteet tai arvot, joita edeltää kaksoispiste tai yhtäsuuruusmerkki, joten ne poistetaan. Tämän toimenpiteen jälkeen jäljelle jääneet viestit ovat raakaversioita lokiviestipohjista. Tässä vaiheessa tavoitteena ei ole saada kaikkia muuttujia vielä poistettua, sillä kaikkien poistaminen vaatisi sovelluksen syvällistä tuntemista.

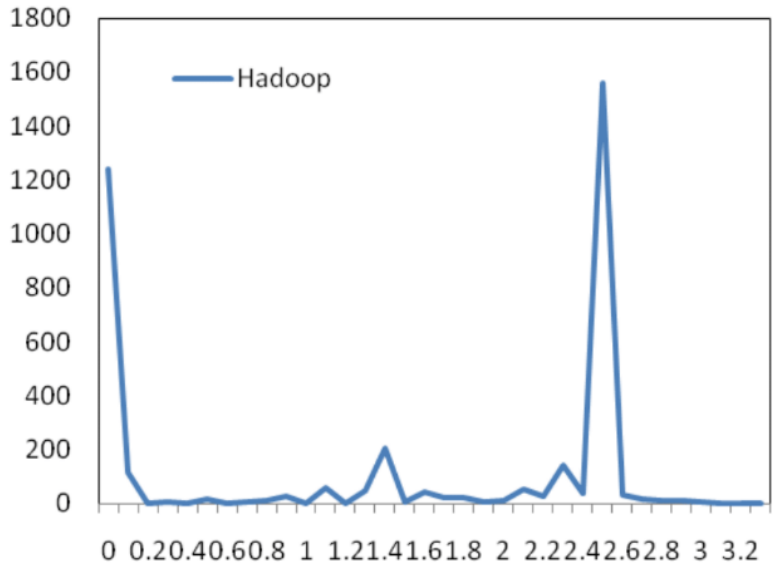
Kuten ryvästämistä esittelevän osion 4.1 yhteydessä esitetään, ryvästämistä varten tulee määritellä, miten kahden käsiteltävän asian etäisyys tai samankaltaisuus mitataan. Tutkijat Fu ja muut (2009) ovat tehneet havainnon, että lokiviestien alun merkkijonot ovat todennäköisemmin vakioon kuuluvia kuin loppupään merkkijonot. Lokiviestien alkupään merkkijonoilla tulisi siis olla suurempi merkitys samankaltaisuuden mittaamisessa kuin loppupään. Lokiviestien väliseksi etäisyyden mitaksi on siksi valittu painotettu tekstin muokkausetäisyys, joka käytännössä tarkoittaa tarvittavien operaatioiden määrää, jotta viesti A saadaan muokattua muotoon B, ottaen huomioon sanojen sijainnin viestissä. Operaatio voi olla kirjaimen lisäys, poisto tai korvaaminen. Formaalisti tarvittavat operaatiot lokiviestipohjille rk_1 ja rk_2 ovat $OA_1, OA_2, \ldots, OA_{EO}$, jossa EO on tarvittavien operaatioiden kokonaismäärä. Painotettu muokkausetäisyys (WED) on formaalisti määriteltynä

$$WED(rk_1, rk_2) = \sum_{i=1}^{EO} \frac{1}{1 + e^{x_i - v}},$$

jossa x_i on i:nnessä operaatiossa OA_i muokattavan sanan indeksi ja v on parametri, joka kontrolloi painotusfunktiota.

Kaksi viestiä kuuluu samaan alustavaan rypääseen, jos niiden välinen etäisyys on pienempi kuin kynnysarvo ς. Kynnysarvon laskemiseksi lasketaan ensin parittainen painotettu muokkausetäisyys jokaiselle raakaversiolle lokiviestipohjasta. Tämän jälkeen käytetään k-keskiarvot-menetelmää (k-means) etäisyyksien ryvästämiseksi kahteen rypääseen. K-keskiarvot-menetelmässä jaetaan datajoukko k:hon rypääseen laskemalla datapisteen etäisyys rypäiden keskipisteistä (Bonaccorso, 2017). Menetelmän tuottamat kaksi ryvästä vastaavat rypäiden sisäisiä ja välisiä etäisyyksiä. Esimerkki painotettujen muokkausetäisyyden jakaumasta on kuvassa 2, jossa muokkausetäisyydet ovat peräisin Hadoopin lokidatasta. Ensimmäinen piikki kuvastaa

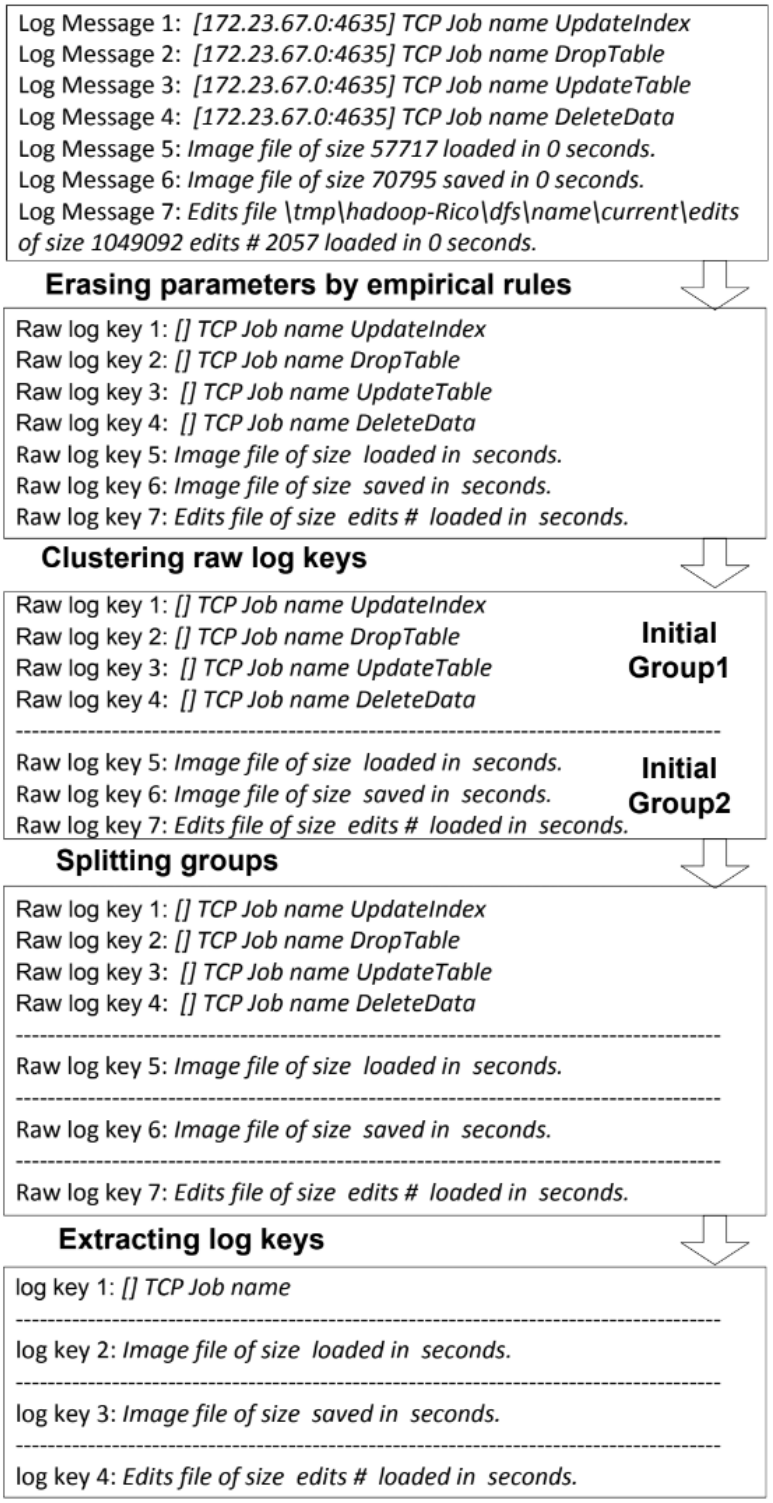
rypäiden sisäisiä etäisyyksiä ja toinen piikki rypäiden välisiä etäisyyksiä. Kuvasta nähtävät kaksi piikkiä osoittavat myös, että käytetty painotettu muokkausetäisyys on hyvä samankaltaisuuden mitta, sillä ryvästen sisäiset etäisyydet ovat pienet, ja ryvästen väliset etäisyydet ovat selkeästi suurempia. Kynnysarvoksi ς valitaan sisäisten etäisyyksien suurin arvo.



Kuva 2: Painotetut muokkausetäisyydet Hadoopissa. X-akselilla ovat etäisyydet ja y-akselilla raakojen parittaisten lokiviestipohjien etäisyyksien määrä (Fu ja muut, 2009).

Tämän jälkeen alustavat rypäät käydään vielä läpi. Rypääseen kuuluvia lokiviestipohjia verrataan toisiinsa ja lasketaan jokaiselle merkkijonon indeksille j montako eri arvoa se saa rypään lokiviestipohjissa. Tätä määrää merkitään VN_j, ja tyypillisesti jos se on suuri, kyseessä on muuttuja ja se voidaan poistaa pohjasta. Jos se on pieni, kyseessä on lokiviestipohjan osa. Eri VN_j:n arvoista etsitään pienin, ja jos se on suurempi tai yhtä suuri kuin määritelty kynnysarvo ϱ, ryvästä ei jaeta. Muussa tapauksessa ryhmä jaetaan pienintä VN_j:n arvoa vastaavaan määrään rypäitä eli jos pienin VN_j on kaksi, ja kynnysarvo ϱ on neljä, jaetaan ryhmä kahteen siten, että samassa ryhmässä ovat lokiviestipohjat, joilla on sama sisältö kohdassa j. Tätä toistetaan kunnes ryhmissä ei ole enää kynnysarvon alittavia toisistaan eriäviä merkkijonoja. Koko edellä mainittu prosessi lokimallien määrittämiseksi on kuvassa 3. Uusille lokiviesteille löydetään vastaava lokiviestipohja poistamalla ensin ilmeiset muuttujat ja tämän jälkeen laskemalla painotettu muokkausetäisyys rypäisiin ja valitaan niistä pienin. Jos etäisyys on suurempi kuin harjoituslokiviestien suurin etäisyys lokiviestipohjiin, uusi lokiviesti merkitään virheviestiksi. Jos etäisyys on pienempi, katsotaan että uuden lokiviestin lokiviestipohja on kyseinen lähin ryväs.

Toisen lokiviestien ryvästämiseen tarkoitetun menetelmän pohjalta on luotu SLCT-työkalu (Simple Logfile Clustering Tool) käyttämä menetelmä (Vaarandi, 2003). Me-



Kuva 3: Lokiviestipohjan jäsentäminen lokitiedostosta (Fu ja muut, 2009).

netelmässä jokainen rivi lokitiedostossa mielletään datapisteenä, jolla on n ulottuvuutta. Datapisteen attribuutti on yksi lokiviestissä esiintyvä sana ja n on maksimi lokiviestin sanojen määrälle. Datapisteet eli lokiviestit muodostavat data-avaruuden. Alue S on osajoukko data-avaruudesta, ja sisältää sellaiset datapisteet, joiden attribuuttien $i_1 \dots i_k$ arvot $v_1 \dots v_k$ ovat identtiset. Formaalisti

$$\{x_{i1} = v_1, \dots, x_{ik} = v_k \mid x \in S\},$$

jossa $1 \leq k \leq n$. Joukkoa $\{(i_1, v_1), \dots, (i_k, v_k)\}$ kutsutaan S:n kiinteiden attribuuttien joukoksi. Jos $k = 1$, kutsutaan aluetta 1-alueeksi, ja tiheä alue sisältää vähintään N datapistettä ja N on käyttäjän määrittelemä kynnysarvo. Toisin sanoen lokiviesteistä etsitään samassa indeksissä esiintyviä sanoja ja muodostetaan niistä monikko (i_1, v_1), jossa i_k on indeksi, jossa sana esiintyy ja v_k on itse sana. Kaikista lokiviestien välisistä yhteisistä sanoista samoissa indekseissä muodostetaan kiinteiden attribuuttien joukko. Jos yhteisiä sanoja on vain yksi, kyseessä on 1-alue. Jos sana esiintyy vähintään N:ssä lokiviestissä, se on tiheä. Esimerkiksi jos lokitiedostossa on lokiviestit 'User admin logged in' ja 'User xyz logged in' niin niistä voidaan muodostaa joukko {(1, 'User'), (3, 'logged'), (4, 'in')}. Viestit muodostavat myös alueen S. Tässä $k = 3$, joten kyseessä ei ole 1-alue, mutta jos kynnysarvo $N = 2$, kyseessä on tiheä alue.

Esitelty algoritmi koostuu kahdesta datan läpikäynnistä (Vaarandi, 2003). Ensimmäisen läpikäynnin tuloksena on yhteenveto datasta ja toisen osan lopputuloksena ryväsehdokkaat. Algoritmin viimeisessä osassa käydään läpi ryväsehdokkaat ja valitaan lopulliset rypäät. Ensin tunnistetaan kaikki tiiviit 1-alueet eli usein esiintyvät sanat. Tämän jälkeen muodostetaan kaikki ryväsehdokkaat, jotka kootaan ehdokastauluun. Dataa käydään läpi viesti viestiltä, ja jos rivillä esiintyvä sana löytyy yhdestä tai useammasta tiiviistä 1-alueesta, siitä muodostetaan ryväsehdokas. Jos kyseistä ehdokasta ei löytynyt vielä ehdokastaulusta, se lisätään siihen ja määritetään tueksi 1. Jos se löytyi jo taulusta, tuen määrää kasvatetaan yhdellä. Klusteriehdokkaaseen kuuluu vain ne sanat, jotka löytyvät 1-alueista eli jos lokiviesti on 'Connection to database established' ja (1, 'Connection') ja (4, 'established') ovat tiiviitä 1-alueita, ryväsehdokas on {(1, 'Connection'), (4, 'established')}. Lopulta ehdokastaulusta voidaan etsiä kaikki ehdokkaat, joiden tuki on suurempi kuin käyttäjän määrittelemä kynnysarvo, ja ne muodostavat rypäät. Yksi ryväs vastaa tiettyä rivihahmoa (line pattern) eli lokiviestipohjaa, esimerkiksi {(1, 'Authentication'), (2, 'successful'), (3, 'user'), (4, 'logged'), (5, 'in')} vastaa rivihahmoa 'Authentication successful, user * logged in.'

Lokiviestipohjat voi muodostaa myös lähdekoodin perusteella. Lähdekoodia ja lokitiedostoja tutkimalla huomattiin, että lokiviestit ovat itseasiassa enemmän struktuuroituja kuin miltä ne ensin vaikuttavat (Xu ja muut, 2009a). Lokiviestien skeema on nähtävissä järjestelmän lähdekoodista, ja se voidaan jäsentää siitä automaattisesti, sillä lokiviestien tulostuslausekkeista on helposti erotettavissa eri lokiviestin muuttujat ja vakiot. Lokiviestipohjien jäsentäminen lähdekoodista tekee menetelmästä ohjelmointikielestä riippuvaisen, mutta helpottaa myös pohjien muodostamista.

Lokiviestin rakenne voidaan saada lokien tulostuslausekkeista melko helposti, sillä monissa ohjelmointikielissä merkkijonoissa esiintyy paikkamerkkejä, jotka korvataan muuttujan arvolla muotoiluvaiheessa. Esimerkiksi C-ohjelmointikielen tapauksessa voisi olla tulostuslauseke `printf(LOG,"starting: database %s in machine %s is %s.", "software_db", gethostname(), "INITIALIZING TABLES")`. Lokituslausekkeesta voidaan tunnistaa helposti, että se sisältää kolme muuttujaa. Muuttujista kaksi ovat tunnisteita ja kolmas tilamuuttuja. Tulostuslausekkeesta näkee suoraan myös muuttujien tyypit, sillä `"%s"` merkitsee merkkijonoa. Jos lokiviestissä olisi kokonaisluku, se merkittäisiin paikkamerkillä `"%d"`. Olio-ohjelmoinnin suhteen lokiviestin rakenteen selvittäminen ei ole yhtä helppoa. Esimerkiksi Javassa luokkahierarkiat ja periminen tuovat monimutkaisuutta lisää, eikä välttämättä ole selvää, mitä tulostuslauseke todellisuudessa tulostaa, jos jokin muuttujista on olio (Xu ja muut, 2009a).

Lähdekoodiperustaisessa jäsentämismenetelmässä syötteenä toimii järjestelmän lähdekoodi ja lokituksen hoitavan luokan nimi, jos se on tiedossa ja sama järjestelmän laajuisesti (Xu ja muut, 2009a). Menetelmässä käytetään olio-ohjelmointikielen tapauksessa abstraktia syntaksipuuta, joka mahdollistaa koodin läpikäynnin ja analysoimisen ja jonka tuottamiseen on olemassa valmiit työkalut. Abstraktia syntaksipuuta käyttäen käydään läpi `toString-metodit` ja lopputuloksena saadaan täydelliset viestipohjat.

Tämän jälkeen käydään vielä lokitiedoston lokiviestit läpi. Tätä varten lähdekoodista eristetyistä viestipohjista luodaan käänteinen indeksi, jonka avulla lokiviestille löydetään nopeasti vastaava lokiviestipohja. Lokiviesteistä poistetaan ensin kaikki numerot ja erikoismerkit ja tämän jälkeen suoritetaan haku käänteiseen indeksiin. Se palauttaa relevanssin perusteella järjestetyn listan kandidaatteja viestipohjista ja niistä valitaan ensimmäinen. Lokiviestipohjan haku indeksistä saadaan helposti hajautettua osittamalla lokitiedosto ja kopioimalla indeksi kaikille työläissolmulle (worker node) (Xu ja muut, 2009a). Osiossa 4.4 esitettyä menetelmää lokitiedoston lokiviestit korvataan pelkän lokiviestipohjan sijaan monikolla $[T(m), K(m), PV(m,1), PV(m,2), \ldots, PV(m, PN(n))]$, jossa T_m on lokiviestin aikaleima, $K(m)$ lokiviestipohja, $PN(m)$ parametrien määrä ja $PV(m,i)$ i:nnes parametri (Lou, Fu, Yang, Xu, & Li, 2010).

3.4 Lokiviestipohjien jatkoprosessointi

Ennen vektorisointia lokiviestipohjia voidaan edelleen prosessoida. Invarianttimenetelmää varten lokiviestit ryhmitellään osiossa 4.4 esitellyn yhteisgeneettisyyden perusteella. Jokaisesta ryhmästä lasketaan lokiviestien määrä jokaiselle eri lokiviestityypille ja muodostetaan tästä viestimäärävektori. Saman järjestelmämuuttujan sisältävät ryhmät muodostavat joukon viestimäärävektoreita ja edelleen viestimäärämatriisin (Lou ja muut, 2010).

Tapahtumiin voidaan lisätä myös painotukset ennen vektorisointia. Tämä perustuu siihen, että toiset viestit sisältävät enemmän ongelmien havaitsemisen kannalta tär-

keää informaatioita kuin toiset (Lin, Zhang, Lou, Zhang, & Chen, 2016). Kun lokeissa esiintyvät tapahtumat on jaoteltu tunnisteiden mukaan lokijaksoihin, lokijaksoon kuuluvan tapahtuman x painoarvo lasketaan seuraavasti:

$$w(x) = 0{,}5 \cdot Norm(w_{idf}(x)) + 0{,}5 \cdot w_{con}(x).$$

Funktio $w_{idf}(x)$ on käänteinen dokumenttitiheys (Inverse Document Frequency), jonka avulla harvoin esiintyville tapahtumille suurempi paino kuin usein tapahtuville. Käänteinen dokumenttitiheys lasketaan

$$w_{idf}(x) = \log(\frac{N}{n_x}),$$

jossa N on lokiviestijaksojen määrä ja n_x niiden jaksojen määrä, joissa viesti x esiintyy. Norm on normalisointifunktio, joka normalisoi käänteisen dokumenttitiheyden avulla painotetun tapahtuman välille 0 ja 1. Normalisointifunktiona käytetään sigmoid-funktiota. Toinen painotusfunktio w_{con} on määritelty

$$w_{con}(x) = \begin{cases} 1 & \text{jos } x \text{ esiintyy joukossa } \Delta\text{X} \\ 0 & \text{muuten} \end{cases}$$

jossa ΔX merkitsee joukkoa, joka sisältää vain tuotannossa esiintyvät tapahtumat. Perustelu tälle painotukselle on se, että tuotannossa esiintyvät lokiviestit ovat todennäköisemmin virhelokiviestejä (Lin ja muut, 2016). Käytettävä data sisältää sekä testi- että tuotantoympäristön lokitiedostoja.

3.5 Lokiviestien jakaminen lokisarjoihin

Kun lokiviestipohjat on muodostettu lähdekoodista tai lokitiedostoista, lokitiedostojen rivien perusteella voidaan muodostaa lokiviestisarjoja. Lokiviestisarjat ryhmitellään tyypillisesti aikaleimojen tai tunnisteiden perusteella. Aikaleimaperusteisesti muodostetut sarjat voidaan jakaa kahteen tyyppiin: kiinteisiin ikkunoihin ja liukuviin ikkunoihin. Kiinteän ikkunan kokoa mitataan aikana, ja se voi olla esimerkiksi tunti tai päivä. Ikkunan koko perustuu lokiviesteissä esiintyviin aikaleimoihin. Kokoa merkitään Δt, ja se määrittää ikkunoiden määrän. Lokit, jotka esiintyvät samassa ikkunassa, kuuluvat samaan lokisarjaan (S. He ja muut, 2016).

Myös liukuvilla ikkunoilla on koko Δt, mutta sen lisäksi myös askelkoko, joka määrittää, kuinka paljon ikkunaa siirretään. Yleensä se on pienempi kuin ikkunan koko, joten ikkunat ovat osittain päällekkäisiä. Myös liukuvien ikkunoiden tapauksessa samassa ikkunassa esiintyvät lokit kuuluvat samaan lokisarjaan, mutta yksi lokiviesti voi tässä tapauksessa kuulua useampaan lokisarjaan (S. He ja muut, 2016).

Suuressa osassa lokiviestejä esiintyy tilamuuttujia, joiden suhteellinen osuus pysyy yleensä melko samana aikaikkunoiden välillä (Xu ja muut, 2009a). Esimerkiksi jokaista tiedoston avaamista merkitsevää tilamuuttujaa kohti on yleensä sama määrä tiedoston sulkevia tilamuuttujia. Tämän suhdeluvun muuttuminen voisi indikoida

ongelmasta esimerkiksi tiedoston sulkemisessa. Tämän tiedon perusteella voidaan muodostaa tilasuhdevektorit $\boldsymbol{y}$, joista jokainen esittää ryhmän tilamuuttujia aikaikkunan sisällä. Jokainen vektorin ulottuvuus esittää yhtä tilamuuttujaa, ja jokaisen ulottuvuuden arvo esittää, kuinka monta kertaa tila esiintyy aikaikkunan sisällä. Aikaikkunan koko on valittu artikkelissa (Xu ja muut, 2009a) automaattisesti niin, että jokainen muuttuja esiintyy vähintään $10D$ kertaa 80 prosentissa kaikista aikaikkunoista. Tässä D on eri tilamuuttujien määrä. Tilasuhdevektoreilla on n ulottuvuutta ja tilasuhdevektoreista m:ssä aikaikkunassa koostetaan $m \times n$ matriisi $\boldsymbol{Y}^S$.

Aikamääreenä voi toimia myös solmutunti (nodehour). Solmutunnilla tarkoitetaan yhtä tuntia yhdellä solmulla, eli lokiviestit erotellaan solmutunneissa paitsi aikaikkunoittain myös solmuittain (Stearley & Oliner, 2008). Solmutunteja käyttävässä menetelmässä erotellaan kokonaisista vakioista välilyönnillä erotetut merkkijonot. Niistä muodostetaan sekä sanoja että termejä, joiden ero on se, että termissä on itse merkkijonon lisäksi tieto sanan paikasta lauseessa. Niinpä eri termeiksi lasketaan esimerkiksi '002STARTING' ja '006STARTING'. Lokeista muodostetaan kaksi $N \times M$ matriisia, joiden nollasta poikkeavat arvot merkitsevät kuinka monta kertaa merkkijono i (rivit) on esiintynyt solmutunnin j (sarakkeet) aikana. Toisessa matriisissa siis riveinä ovat sanat ja toisessa termit.

Lokiviestisarjat voidaan muodostaa ajan lisäksi myös tunnisteiden perusteella. Tunnisteiden perusteella voidaan tunnistaa suorituspolkuja. Esimerkiksi lohkotunniste voi esiintyä lokiviesteissä kun lohkolle varataan muistia, kun siihen kirjoitetaan, se poistetaan tai replikoidaan. Jokaisessa lokisarjassa esitetään siis yhteen tunnisteeseen liittyvät lokiviestit (S. He ja muut, 2016).

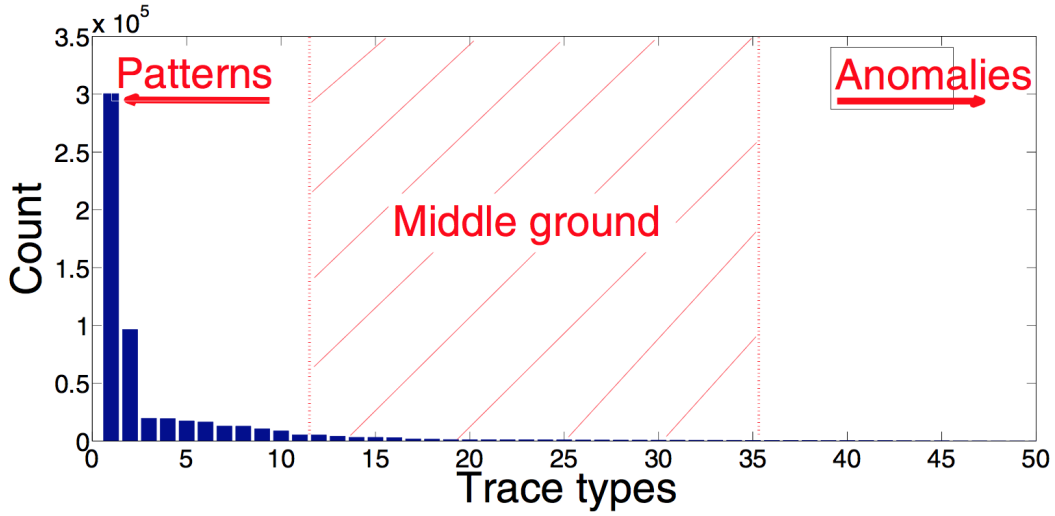
Lokiviesteissä esiintyy paljon tunnisteita ja ryhmittelemällä viestit, joissa esiintyy sama tunniste, muodostetaan viestimäärävektorit (Xu ja muut, 2009a). Ryhmittely suoritetaan Viestimäärävektorien muodostaminen on esitetty algoritmissa 1 (Xu ja muut, 2009a). Yhteen tunnisteeseen liittyy siis yksi viestiryhmä, ja viestimäärävektorin arvot ilmaisevat minkä tyyppisiä viestejä ryhmässä on, kun jokainen vektorin ulottuvuus esittää yhtä lokiviestityyppiä. Yksi vektori esittää yhteen tunnisteeseen liittyvää lokiviestiryhmää. Tunniste on todennäköisesti objektin, esimerkiksi transaktion tunniste, jos se esiintyy useasti, sillä on useita uniikkeja arvoja ja se esiintyy usean tyyppisissä lokiviesteissä. Tunnisteiden uniikkien arvojen määrä on huomattavasti laajempi kuin tilamuuttujissa, joten tunnistetta kuvaavat muuttujat on helppo tunnistaa sen perusteella. Viestimäärävektoreista muodostetaan $m \times n$ matriisi $\boldsymbol{Y}^m$, jossa jokainen rivi on viestimäärävektori y. Matriisissa $\boldsymbol{Y}^m$ on n saraketta, jotka kuvaavat viestityyppejä, ja m riviä, jotka kuvaavat tunnisteisiin liittyviä viestiryhmiä. Lokiviestin tyypillä tarkoitetaan siihen liittyvää lokiviestipohjaa.

Käynnissä olevan (online) järjestelmän anomalioiden havaitsemiseen liittyy omanlaisiaan haasteita verrattuna siihen, että lokeja analysoitaisiin poiskytketystä (offline) järjestelmästä. Koko suorituspolkua ei ole välttämättä heti saatavilla eikä voida tietää, tuleeko suorituspolun päättävä tapahtuma koskaan. Tapahtumaa ei kuitenkaan voida jäädä odottamaan ikuisesti, ja odotusaika voikin dominoida lopullista suoritusaikaa. Toisaalta jos taas anomalian tunnistus suoritetaan heti ensimmäisen

Algoritmi 1 Viestimäärävektorin luominen

1. Etsi lokeista kaikki viestimuuttujat, joilla on seuraavat ominaisuudet:
   a. Muuttuja esiintyy monta kertaa;
   b. Muuttujalla on monta uniikkia arvoa;
   c. Muuttuja esiintyy useassa viestityypissä.
2. Ryhmittele viestit yllä löydettyjen muuttuja-arvojen eli todennäköisten tunnisteiden perusteella.
3. Jokaiselle viestiryhmälle, luo viestimäärävektori $y = [y_1, y_2, \ldots, y_n]$, jossa y_i on viestityypin i määrä viestiryhmissä.

lokitapahtuman ilmetessä, ei havaitsemisessa voida soveltaa hahmojen tunnistamista. Ratkaistakseen tämän ongelman tutkijat Xu, Huang, Fox, Patterson, ja Jordan (2009b) käyttävät kaksivaiheista menetelmää. Kaksivaiheisen tunnistuksen ensimmäisessä vaiheessa karsitaan yleiset, hahmoihin sopivat tapahtumat ja suoritetaan pääkomponenttianalyysi vain tapahtumille, jotka eivät ole dominoivia, mutta joista ei voi olla varma, ovatko ne anomalioita. Nämä tapahtuvat näkyvät kuvassa 4 keskellä. Ensimmäinen vaihe varmistaa, että tieto anomalioista saadaan järkevän ajan sisällä määrittämällä maksimiviive anomalioiden havaitsemiselle, ja pääkomponenttianalyysi parantaa havaitsemistarkkuutta tunnistamalla epävarmoista tapauksista normaalit tapahtumat (Xu ja muut, 2009b). Pääkomponenttianalyysin toiminnasta kerrotaan enemmän osiossa 4.3.



Kuva 4: Hahmojen esiintymisen jakauma, jossa eroteltuna normaalit hahmot, anomaliat ja epävarmat tapaukset (Xu ja muut, 2009b).

Kaksivaiheisessa menetelmässä tapahtumaketju on määritetty ryhmänä tapahtumia, joissa on sama muuttuja, ja sessio on tapahtumaketjun alijoukko, jossa olevat tapahtuvat liittyvät toisiinsa ja jolla on ennustettava kesto. Menetelmä jakaa lokisarjat sessioihin (Xu ja muut, 2009b). Toistuvien hahmojen louhimisen tarkoituksena on paitsi jakaa lokisarjat sessioihin, myös mitata sessioiden kesto. Session kesto mitataan viimeisen ja ensimmäisen aikaleiman välisenä erotuksena. Toistuvalla hahmolla viitataan sessioon ja sen kestojakaumaan kun pätee, että sama sessio esiintyy useissa tapahtumaketjuissa ja että session kesto mahtuu lähes täysin käyttäjän määrittelemän havaitsemisviiveen T_{max} sisään. Havaitsemisviive T_{max} on käyttäjän määrittelemä yläraja sille, kuinka nopeasti anomaliat tulee tunnistaa.

Toistuvien hahmojen etsiminen on kolmiosainen prosessi. Ensin jokaisesta suorituspolusta etsitään tapahtuma, jonka jälkeinen aikaväli on kymmenen kertaa yhtä pitkä kuin aika, joka on kulunut suorituspolun alusta. Löydettyä tapahtumaa edeltävät tapahtumat muodostavat session, joka esitetään viestimäärävektorina. Tämän jälkeen tunnistetaan dominoiva hahmo etsimällä ensin sessioiden mediodi, jonka etäisyys on pienin kaikkiin muihin vektoreihin. Sen jälkeen tarkastetaan, että sessio esiintyy vähintään $0,2M$ osuudessa tapahtumaketjuista, missä M on tapahtumaketjujen määrä. Jos session medoidi ei täytä tätä minimituen ehtoa, valitaan sessiosta seuraavaksi medoidia lähinnä oleva piste, joka täyttää minimituen ehdon.

Toistuvien hahmojen etsimisen viimeisessä vaiheessa haetaan alkuperäisestä datasta kaikki tapahtumat, jotka vastaavat löydettyä toistuvaa hahmoa ja estimoidaan tapahtumien kestojakaumaa. Jakauman avulla voidaan laskea katkaisuaika (cut-off time) $T_{katkaisu}$, joka ilmaisee missä ajassa 99,95 prosenttia sessioista pitäisi päättyä. Osa suoritettavista operaatioista on hyvin pitkäkestoisia, joten $T_{katkaisu}$ saattaa olla hyvinkin pitkä. Jos $T_{katkaisu} > T_{max}$, hahmoa ei huomioida havaitsemisvaiheessa. Koska suoritusten jakaumalla on pitkä häntä, artikkelissa on käytetty potenssilakijakaumaa mallin muodostamiseen. Alkuperäisestä datasta poistetaan katkaisuajan laskemisen jälkeen kaikki toistuvaan hahmoon täsmäävät tapahtumat ja toistuvien hahmojen etsimistä toistetaan kunnes datassa ei ole jäljellä hahmoja, jotka täyttävät minimituen ehdon. Viimeiseksi toistuvien hahmojen etsimisen jälkeen käytetään pääkomponenttianalyysia anomalioiden tunnistamiseen.

4 Koneoppimismenetelmät

Kun lokiviesteistä on muodostettu mallit ja niiden perusteella on muodostettu vektorit ja edelleen matriisit, voidaan niistä löytää mielenkiintoiset hahmot koneoppimisen avulla. Esiteltävät koneoppimismenetelmät on kuvattu lyhyesti taulukossa 2.

Nimi	Kuvaus
Ryvästäminen	Ryvästämisellä tarkoitetaan menetelmää, jossa datajoukko jaetaan samankaltaisuuden tai etäisyyden perusteella pienempiin ryhmiin eli rypäisiin. Ryvästämistä varten tulee määritellä kahden datapisteen ja kahden rypään etäisyyden tai samankaltaisuuden mitta. Esimerkki ohjaamattomasta ryvästämismenetelmästä on k-keskiarvot (k-means).
Painotusmenetelmät	Painotusmenetelmillä pyritään korostamaan datapisteiden jotain tärkeänä pidettävää ominaisuutta kuten entropiaa. Entropialla tarkoitetaan informaatioteoriassa kaikkien datapisteiden yhteenlaskettua informaation odotusarvoa.
Pääkomponenttianalyysi	Pääkomponenttianalyysi pyrkii erottamaan datajoukosta merkityksellisimmät piirteet eli pääkomponentit. Pääkomponentit pystyvät selittämään suurimman osan datajoukon vaihtelusta pienemmässä määrässä ulottuvuuksia kuin alkuperäisessä datassa.
Invariantit	Invariantti on predikaatti, joka kuvaa järjestelmän suorituspolun yhtälöinä tai vektorina θ. Invariantit louhitaan datapisteiden muodostamasta matriisista $\mathbf{X}$ etsimällä harvoja, kompakteja ja kokonaisluvuista muodostuvia vektoreita. Suorituspolut, jotka rikkovat löydettyjä invariantteja tulkitaan anomalioiksi.

Taulukko 2: Esiteltävät koneoppimismenetelmät.

4.1 Ryvästäminen

Ryvästämisellä tarkoitetaan datapisteiden $X = \{x_1, x_2, \ldots x_n\}$ jakamista ryhmiin jonkin kriteerin $G(x_i)$ perusteella. Näitä ryhmiä kutsutaan rypäiksi ja rypäisiin jakamista ryvästämiseksi. Kriteeri $G(x_i)$ määrittää datapisteen kuulumisen ryväkseen ja vaihtelee datatyypeittäin ja ryvästysmenetelmittäin. Ryvästysmenetelmät voidaan jakaa karkeasti koviin ja pehmeisiin ryvästysmenetelmiin. Kovat ryvästysmenetelmät tarkoittavat sitä, että datapiste voi kuulua vain yhteen rypääseen, kun taas pehmeät ryvästysmenetelmät määrittävät, kuinka sopiva datapiste on tiettyyn rypääseen. Pehmeiden ryvästysmenetelmien tuloksena on jaukauma

$$m_i = \{(m_i^0, m_i^1, \ldots, m_i^t) | m_i^k \in [0, 1]\},$$

jossa vektori m_i esittää datapisteen x_i sopivuutta rypäisiin, ja se voidaan normalisoida todennäköisyysjakaumaksi (Bonaccorso, 2017).

Hierarkkinen ryvästys pyrkii tunnistamaan osittaisten rypäiden keskinäisen hierarkian. Se voidaan jakaa kahteen erilaiseen ryvästysmenetelmään: kasautuvaan (ag-

glomerative) ja jakavaan (divisive) ryvästämiseen. Kasautuvalla hierarkisella ryvästyksellä tarkoitetaan ryvästysmenetelmää, jossa jokainen datapiste on ensin oma rypäänsä. Rypäitä aletaan iteratiivisesti sitten yhdistelemään niin, että rypäät, joiden etäisyys toisistaan on pienin, yhdistetään. Tätä jatketaan kunnes määritelty lopetuskriteeri on täytetty. Jakava ryvästys aloittaa päinvastaisesta tilanteesta: kaikki datapisteet kuuluvat yhteen rypääseen ja iteratiivisesti edeten luodaan uusia, pienempiä rypäitä. Rypäitä jaetaan niiden eriävyyden perusteella.

Kahden datapisteen etäisyyden mittana voi toimia esimerkiksi euklidinen etäisyys

$$d_{euklidinen}(x_1, x_2) = \|x_1 - x_2\|_2 = \sqrt{\sum_i (x_1^i - x_2^i)^2},$$

Manhattan-etäisyys

$$d_{manhattan}(x_1, x_2) = \|x_1 - x_2\|_1 = \sum_i |x_1^i - x_2^i|$$

tai kosinietäisyys

$$d_{kosini}(x_1, x_2) = 1 - \frac{x_1 \cdot x_2}{\|x_1\|_2 \|x_2\|_2}.$$

Samankaltaisuuden mitan lisäksi määritellään yhdistämisehto. Yhdistäminen voidaan suorittaa esimerkiksi kauimman naapurin etäisyyden perusteella tai aritmeettisen keskiarvon perusteella. Kauimman naapurin etäisyyden perusteella ryvästämisessä yhdistetään rypäät, joiden kauimmaisten jäsenten etäisyys on pienin. Formaalisti esitettynä siis kahden rypään välinen etäisyys on tässä tapauksessa

$$L_{ij} = max\{d(x_a, x_b) \mid x_a \in C_i, x_b \in C_j\}.$$

Aritmeettisen keskiarvoon perustuvassa ryvästämisessä rypäiden yhdistäminen perustuu nimen mukaisesti rypäiden elementtien etäisyyksien keskiarvoon . Formaalisti

$$L_{ij} = \frac{1}{|C_i||C_j|} \sum_{x_a \in C_i} \sum_{x_b \in C_j} (d(x_a, x_b)).$$

Ne rypäät, joiden etäisyys näillä menetelmillä mitattuna on pienin, yhdistetään (Bonaccorso, 2017).

4.1.1 Ryvästämisen sovelluksia

LogCluster menetelmässä oli valittu ryvästämiseen kasaantuva hierarkinen ryvästysmenetelmä (Lin ja muut, 2016). Vektorien etäisyyden laskemiseen käytetään kosinietäisyyttä, ja rypäiden välisenä etäisyyden mittana käytetään kauimman naapurin etäisyyttä. Ryvästämisalgoritmin suoritus lopetetaan, kun kahden rypään välinen etäisyys ylittää määritetyn kynnysarvon θ.

Ryvästämisen jälkeen lasketaan vielä jokaiselle rypäälle edustuksellisin viestijakso. Se tehdään käymällä läpi rypään jäsenet ja laskemalla jokaiselle niistä edustuksellisuus. Edustuksellisuus i:nnelle viestijaksolle lasketaan seuraavasti:

$$Score(i) = \frac{1}{n-1} \sum_{j=1}^{n} (1 - Similarity(x_i, x_j)),$$

jossa n on rypään jäsenten määrä. Rypään edustavin viestijakso on se, jonka $Score(i)$ on pienin. Edustuksellisimmat viestijaksot ovat ehdokkaita manuaalisesti läpikäytäviksi viestijaksoiksi.

Jos käytössä on tietoa funktioiden suorituksen aloituksesta ja lopetuksesta, ryvästämistä varten voidaan muodostaa funktioprofiileja (Mirgorodskiy ja muut, 2006). Polkukohtainen funktioprofiili on määritelty vektorina

$$p(h) = \Big(\frac{t(h, f_1)}{T(h)}, \ldots, \frac{t(h, f_F)}{T(h)} \Big),$$

jossa F on järjestelmän funktioiden kokonaismäärä, ja h on solmu, jolla funktio suoritetaan. Vektorin i:nnes komponentti vastaa funktiota f_i ja esittää ajan $t(h, f_i)$, joka vietetään kyseisessä funktiossa suhteessa järjestelmän kokonaissuoritusaikaan $T(h) = \sum_{i=1}^{F} (t(h, f_i))$. Kahden solmun g ja h tuottamien suorituspolkujen välinen etäisyys on määritelty komponenttikohtaisen erovektorin Manhattan-pituutena δ

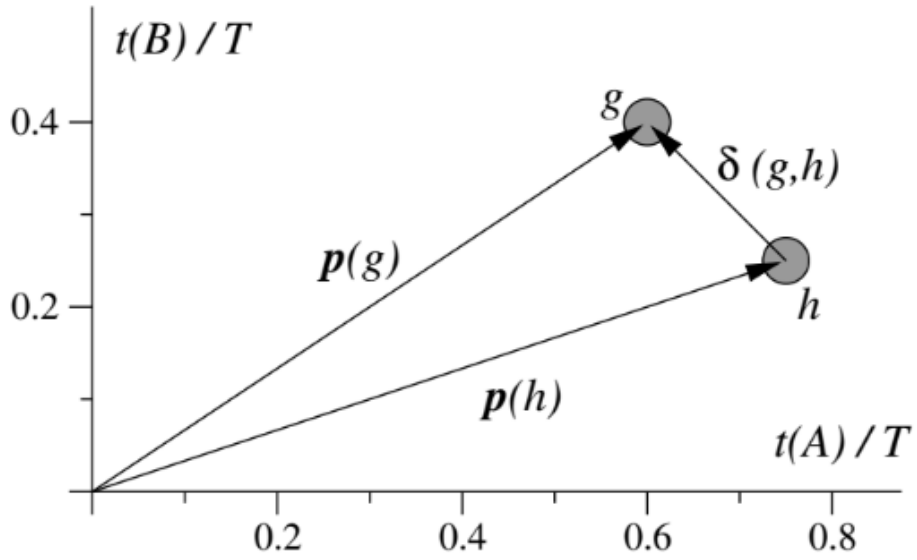
$$d(g, h) = \delta(g, h) = \sum_{i=1}^{F} |\delta_i|,$$

missä δ_i on yksittäisen funktion f_i vaikutus etäisyyteen. Mitä samankaltaisemmin solmut g ja h toimivat, sitä lähempänä ne ovat toisiaan ja sitä pienempi $d(g, h)$ on. Kuva 5 esittää geometrisen tulkinnan profiileista ja niiden välisestä etäisyydesta. Esimerkin järjestelmässä on kaksi funktiota A ja B. Solmu g:n profiili on (0,6, 0,4) ja solmun h profiili (0,75, 0,25).

Etäisyyden määrittelyn jälkeen lasketaan epäilypisteytys (suspect score), joka ilmaisee etäisyyden polun ja normaaliksi uskottujen polkujen joukon välillä. Epäilypisteytys suoritetaan joukolle polkuja T, joiden oletetaan kuvaavan suurimmaksi osaksi normaalia suoritusta ja sisältävän vain pieniltä osin poikkeavuuksia. Poikkeavuudet havaitaan järjestämällä datapisteet sen mukaan, kuinka kaukana ne ovat k:nnesta naapuristaan. Jokaiselle polulle $h \in T$ muodostetaan polkujen jono $T_d(h) = \langle h_1, h_2, \ldots, h_{|T|} \rangle$, johon pätee, että $d(h, h_i) \leq d(h, h_{i+1}), 1 \leq i \leq |T|$. Epäilypisteytys polulle h on määriteltynä sen ja sen k:nneksi lähimmän naapurin etäisyytenä

$$\sigma(h) = d(h, h_k).$$

Sopiva k:n arvo on tärkeä, sillä jos se on liian matala, väärien negatiivisten määrä kasvaa ja jos se puolestaan oli liian korkea, väärien positiivisten määrä kasvaa. Valitulla k:n arvolla on siis myös merkitystä siihen, kuinka herkästi polku tulkitaan



Kuva 5: Suorituspolkuprofiileihin perustuvan poikkeavuuksien havaitsemisen geometrinen esitys (Mirgorodskiy ja muut, 2006).

anomaliaksi. Artikkelin tutkimuksessa todettiin, että menetelmä toimi hyvin k:n arvoilla välillä $(3, |T|/4)$. Jos epäilypisteytys on korkea, polku tulkitaan anomaliaksi, muuten normaaliksi. Menetelmä perustuu siihen, että normaaleilla tapahtumilla on paljon naapureita. Anomalioiksi saattaa näin päätyä myös normaalit, mutta harvinaiset tapahtumat.

Artikkelissa esitettiin myös anomalian tarkempaa paikallistamista helpottava menetelmä. Etsimällä suurimman δ_i voi siis tunnistaa funktion, joka on vaikuttanut eniten polun poikkeavuuteen. Poikkeava funktio af ilmaistaan formaalisti

$$\text{af} = \underset{1 \leq i \leq F}{\text{argmax}} \, |\delta_i|.$$

Tätä voidaan vielä parantaa jakamalla suorituspolut saman pituisiin aikajaksoihin ja tunnistamalla anomaliat kyseisistä ajanjaksoista. Suorituspolku, joka on alkanut ensimmäisenä ja joka on poikkeava, merkitään tutkittavaksi ensimmäisenä (Mirgorodskiy ja muut, 2006).

Aiemmin mainitun ohjelmistokehyksen Pinpointin (Chen ja muut, 2002) viimeinen osa on tuotettujen lokitiedostojen analysointi. Analysointi suoritetaan ryvästämällä ja löytämällä näin todennäköisin komponentti, joka on aiheuttanut virheen. Pinpoint käyttää painottamatonta pari-ryhmä -menetelmää (UPGMA) käyttäen aritmeettista keskiarvoa. Menetelmä on yksinkertainen hierarkkinen ryvästysmenetelmä, jossa ryväs yhdistetään aina lähimpänä olevaan rypääseen käyttäen rypäiden etäisyyden mittana niiden pisteiden keskiarvoja. Datapisteiden etäisyys toisistaan lasketaan

käyttäen Jaccardin samankaltaisuuskerrointa, eli tässä tapauksessa komponenttien suhteellista esiintymistä pyynnöissä.

4.2 Painotusmenetelmät

Anomalioiden havaitsemiseen on käytetty myös painotusmenetelmiä, joiden avulla ajanjakso luokitellaan joko virheen sisältäväksi tai virheettömäksi (Stearley & Oliner, 2008; Oliner, Aiken, & Stearley, 2008). Ajanjaksosta käytetään termiä solmutunti ja se tarkoittaa tuntia, jonka aikana yksi solmu on tuottanut lokia.

Pohjana artikkelissa (Stearley & Oliner, 2008) tutkituille menetelmille on matriisin $\mathbf{X}$ j:nnen sarakkeen merkitsevyysfunktio $|x_j| = \sqrt{\sum_{i=1}^{m} x(i,j)^2}$, missä m on matriisin rivien määrä. Merkitsevyysfunktioon on tutkittu kahta eri painotusmenetelmää, joista toinen pohjautuu käänteiseen dokumenttitiheyteen ja toinen mittaa entropiaa. Käänteisen dokumenttitiheyden avulla tehty painotus on $g(i) = \log_2(1 + \frac{n}{df_i})$, jossa df_i merkitsee niiden solmutuntien määrää, joissa sanaa ja sen indeksiä esittävä termi i esiintyy ja n on sarakkeiden määrä. Merkitsevyysfunktio muuttuu siis muotoon

$$|x_j| = \sqrt{\sum_{i=1}^{m} (g(i)x(i,j))^2}.$$

Tämän tarkoituksena on painottaa niitä termejä enemmän, jotka ovat hyödyllisiä solmutuntien välillä eron tekemisessä. Toinen käytetty menetelmä on datan entropiaa mittaava menetelmä. Tässä

$$g(i) = 1 + \frac{1}{\log_2 n} \sum_{j=1}^{n} p_{ij} \log_2(p_{ij}),$$

jossa $p_{ij} = \frac{x(i,j)}{\sum_{j=1}^{n} x(i,j)}$ kuvaa termin i suhteellista esiintymistä solmutunnissa j. Tämä johtaa siihen, että yhtä monta kertaa eri solmutunneissa esiintyvät termit saavat painon 0. Näin voidaan erotella termeistä ne, joiden esiintyminen on johdonmukaista ja jotka eivät todennäköisesti ole anomalioita. Jos termi esiintyy kaikissa solmutunneissa, mutta sen esiintyminen on keskittynyt erityisesti tiettyihin solmutunteihin, termin merkitsevyys voi olla lähellä yhtä. Entropian $g(i)$ laskemisen jälkeen jokaisesta termistä otetaan logaritmi, jolloin merkitsevyysfunktio on

$$|x_j| = \sqrt{\sum_{i=1}^{m} (g(i) \log_2(x(i,j)))^2}.$$

Samoja painotusmenetelmiä sovelletaan kahteen matriisin, jotka sisältävät aggregoitua dataa raakojen solmutuntien sijaan. $\mathbf{Y}$ on solmuaggregoitu $M \times N$ matriisi, jossa sarakkeiden määrä N on sama kuin solmujen kokonaismäärä. Sarake j on sama kuin elementtikohtainen $\mathbf{X}$:n kolumnien summa solmulle j. Toinen muodostettava matriisi $\mathbf{Z}$ on muuten sama, paitsi että se on aika-aggregoitu matriisi ja rivien määrä H vastaa tuntien kokonaismäärää.

Tutkijat ovat esitelleet menetelmää myös toisessa artikkelissaan (Oliner ja muut, 2008). Siinä he keskittyvät nimenomaan solmuaggregoituun dataan ja entropiapainotukseen ja kutsuvat menetelmää Nodeinfoksi. Lopullinen merkitsevyysfunktion arvo on artikkelissa nimeltään Nodeinfoarvo, ja solmutunnit järjestetään sen mukaan. Runsaan informaation termejä sisältävät solmutunnit sijoittuvat korkealle ja ne, jotka sisältävät vain vähän informaatiota, sijoittuvat matalalle. Toisin sanoen järjestyksessä ensimmäiset ovat todennäköisemmin anomalioita kuin ne, jotka ovat solmutuntilistan häntäpäässä.

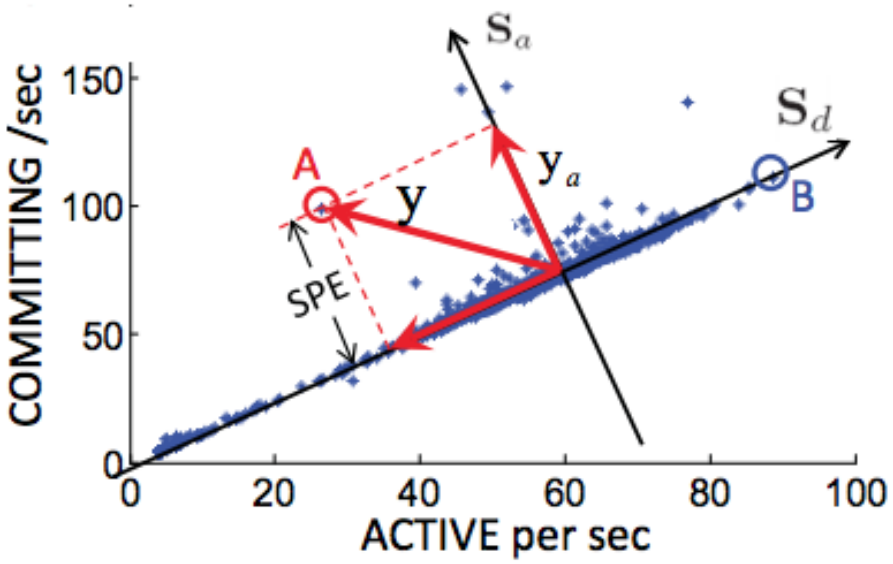
Datan aggregoimisen ja entropian laskemisen jälkeen tutkijat paransivat Nodeinfon suoriutumista siten, että saman tyyppisiin tehtäviin keskittyvät solmut käyttivät yhtä Nodeinfoa sen sijaan, että kaikki olisivat käyttäneet samaa. Parempi suoriutuminen johtuu siitä, että tilastollisissa anomalioiden havaitsemismenetelmissä verrataan otosta viitejakaumaan ja ne mittaavat vaihtelua normaalista. Tästä syystä tilastolliset anomalian havaitsemismenetelmät suoriutuvat paremmin silloin kun viitejakauma on samasta homogeneettisestä populaatiosta kuin otos (Oliner ja muut, 2008).

4.3 Pääkomponenttianalyysi

Pääkomponenttianalyysi (Principal component analysis) on menetelmä, jonka avulla voidaan löytää vahvasti korreloivasta datasta merkittävimmät piirteet projisoimalla se matalaulotteisempaan avaruuteen. Tämä pyritään tekemään niin, että informaation määrä pysyy mahdollisimman lähellä alkuperäistä. Havaintomatriisi X koostuu p:stä piirteestä (sarakkeet) ja n:stä havainnosta. Kaikki p-ulotteisen avaruuden ulottuvuudet eli piirteet eivät ole yhtä mielenkiintoisia (Tibshirani ja muut, 2013). Havaintomatriisi X muodostuu kahden pienen matriisin T ja P tulosta, ja niiden avulla voidaan löytää X:n tärkeimmät hahmot. Piirteen p merkittävyyttä mitataan poikkeavuusvarianssin suuruudella. Havaintomatriisista etsitään siis k pääkomponenttia, jotka selittävät suurimman osan datasta. Anomaliat havaitaan mallissa siitä, että piirteiden korrelaatiot eli piirteiden väliset suhteet muuttuvat (Dunia & Qin, 1997; Wold, Esbensen, & Geladi, 1987).

Koska viestiryhmien viestit ovat vahvasti korreloivia, piirrevektoreiden ulottuvuudet ovat myös hyvin vahvasti korreloivia. Pääkomponenttianalyysin avulla pyritään tunnistamaan epänormaalit vektorit, joiden korrelaatio on poikkeava. Esimerkkikuvassa 6 esitetään kaksiulotteinen tilasuhdevektorien matriisi Y^s, ja kuvassa S_d kuvaa vahvaa korrelaatiota ulottuvuuksien välillä eli normaalia aliavaruutta, ja S_a kuvaa anomalia-aliavaruutta. Pääkomponenttianalyysi tunnistaa siis k-ulotteisen normaalien tapahtumien aliavaruuden S_d alkuperäisestä n-ulotteisesta avaruudesta ja Jäljelle jäävät ulottuvuudet $n - k$ muodostavat anomalia-aliavaruuden. Vektorit kaukana S_d:stä ovat anomalioita, kuten kuvassa piste A.

Anomalioiden tunnistus etäisyyden perusteella voidaan esittää formaalisti neliöllisenä ennustevirheenä (squared prediction error) $SPE \equiv \|\boldsymbol{y}_a\|^2$ eli neliöllisenä y_a:n pituutena. Vektori y_a on vektorin y projektio anomalia-aliavaruuteen S_a, ja se voi-

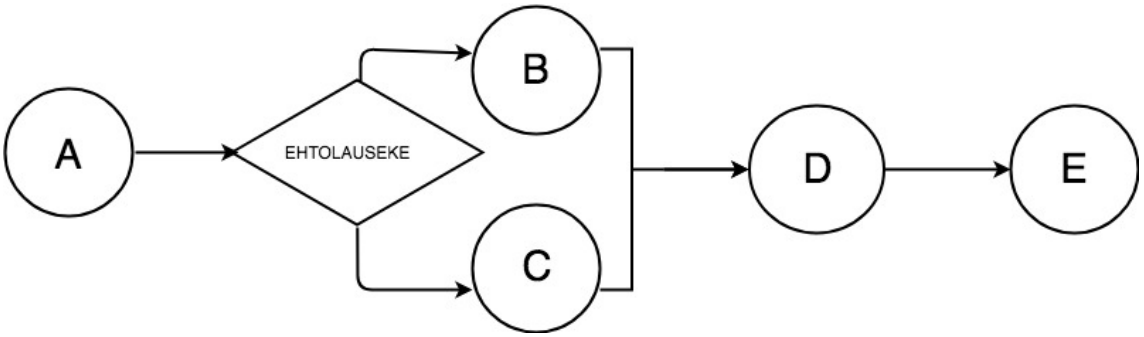


Kuva 6: Esimerkki pääkomponenttianalysin tuottamista aliavaruuksista (Xu ja muut, 2009a).

daan laskea $y_a = (\boldsymbol{I} - \boldsymbol{P}\boldsymbol{P}^T)\boldsymbol{y}$, jossa $\boldsymbol{P} = [\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_k]$ on valittu pääkomponenttianalyysilla ja se esittää ensimmäiset k pääkomponenttia. Vektori y merkitään epänormaaliksi, jos $SPE > Q_\alpha$, jossa Q_α merkitsee kynnysarvoa $(1 - \alpha)$ luottamustasolla. Q:n automaattiseen laskemiseen käytetään Q-tilastoa (Q-statistics) ja se varmistaa, että väärien positiivisten osuus on enintään α olettaen, että havaintomatriisi Y noudattaa moniulotteista normaalijakaumaa.

4.4 Invariantit

Anomalioita voidaan havaita louhimalla lokitiedostoista järjestelmän invariantteja. Koko osio perustuu artikkeliin (Lou ja muut, 2010), jossa invariantit esiteltiin. Järjestelmän invariantilla tarkoitetaan predikaattia, jonka paikkaansapitävyys ei muutu esimerkiksi eri kuormien tai syötteiden johdosta. Invariantteja voi löytää järjestelmien suorituspoluista, ja ne voi esittää yhtälöinä. Esimerkkikuvan 7 suorituspolun osan voisi esittää esimerkiksi c(A)=c(B)+c(C), jossa c(A), c(B) ja c(C) kuvaavat lokiviestien A, B ja C määrää järjestelmän suorituspolkujen vaiheissa. Esimerkki siitä, miltä se voisi näyttää järjestelmän lähdekoodissa on esimerkkikoodissa 2. Suorituspolkuinvarianttien perusteella voidaan tunnistaa anomalioita, sillä ne ilmenevät yleensä poikkeavina suorituspolkuina. Suorituspolut esiintyvät lokitiedostoissa lokiviestijonoina, joten invariantteja louhitaan lokitiedostoista.



Kuva 7: Esimerkki järjestelmän suorituspolusta.

Lineaarinen suhde m:lle erityyppiselle lokiviestille voidaan kirjoittaa

$$a_0 + a_1 x_1 + a_2 x_2 + \cdots + a_m x_m = 0, \tag{1}$$

jossa x_j on niiden lokiviestien määrä, joiden tyyppi-indeksi on j. $\theta = [a_0, a_1, a_2, \cdots, a_m]^T$ on vektori, joka esittää yhtälön kertoimet. Invariantit voidaan siis esittää vektorina θ. Esimerkiksi yllä esitetty yhtälö c(A)=c(B)+c(C) voidaan esittää myös vektorina $\theta = [0, 1, -1, -1, 0, 0]^T$. Jos siis lokiviestiä A on yksi kappale, joko lokiviestiä B tai lokiviestiä C tulee olla myös yksi. Jos lokiviestiä B on yksi kappale, yhtälö (1) on

$$0 + 1 \times 1 + (-1) \times 1 + (-1) \times 0 + 0 + 0 = 0.$$

Jos lokiviestiä B onkin kaksi kappaletta ja lokiviestiä A vain yksi, kyseinen suorituspolku rikkoo tätä invarianttia. Taulukossa 3 näkyy esimerkkejä kuvasta 7 löytyvistä yhtälöistä ja niiden vastaavista kertoimista. Kaikista invarianteista voidaan muodostaa yhtälön (1) mukaisesti joukko yhtälöitä, jotka voidaan ilmaista myös matriisina $\boldsymbol{X}$. Matriisissa sarakkeina ovat lokiviestityypit ja riveinä lokisarjat $L_i,\ i = 1, 2, \ldots, n$, jotka kuvaavat aiempia suorituksia. Solun arvo x_{ij} on kyseisen viestityypin määrä lokisarjoissa. Matriisi $\boldsymbol{X}$ on formaalisti kuvattuna

$$\boldsymbol{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1m} \\ 1 & x_{21} & x_{22} & \ddots & x_{2m} \\ \vdots & \vdots & \vdots & \vdots & \\ 1 & x_{n1} & x_{n_2} & \cdots & x_{nm} \end{bmatrix}$$

Kaikkien invarianttivektorien θ:n tulee siis ratkaista yhtälö

$$\boldsymbol{X}\theta = 0. \tag{2}$$

Yhtälö kuvaa invarianttien peruspiirrettä, mikäli kerätyt lokit eivät sisällä virhetilanteita. Todellisuudessa tämä ei kuitenkaan useinkaan pidä täysin paikkaansa, ja kerätyt lokit sisältävät myös pienissä määrin virhetilanteita. Artikkeli tekee oletuksen, että virheiden ja kohinan määrä on alle 5% ja ratkaisee ongelman minimoimalla $\|\boldsymbol{X}\theta\|_{\mathbf{0}}$ eli invariantin θ rikkovien lokisarjojen määrän.

```
public Long insertData(Input sanitizedInput) {
        //Lokiviesti A
        logger.info("Vastaanotettiin lisättävä data.");
        Input existingInput=database.find(sanitizedInput);
        Long id=null;
        if(existingInput != null) {
                //Lokiviesti B
                logger.info("Input olemassa jo tietokannassa, ei
                 ↪  lisätä uudelleen");
                id=existingInput.getId();
        } else {
                //Lokiviesti C
                logger.info("Lisätään kantaan uusi Input {}",
                 ↪  sanitizedInput);
                id=database.insertInput(sanitizedInput);
        }
        //Lokiviesti D
        logger.info("Input tietokannassa, tunniste: {}", id);
        //Lokiviesti E
        logger.info("Palautetaan tietokantatunniste
         ↪  jatkoprosessointia varten");
        return id;
}
```

Koodiesimerkki 2: Esimerkki ohjelmakoodista, jonka suoritus voidaan esittää kuvassa 7 näkyvänä suorituspolkuna.

Yhtälö	**Kertoimet**
$c(A) = c(B) + c(C)$	[0, 1, -1, -1, 0, 0]
$c(A) = c(E)$	[0, 1, 0, 0, 0, -1]
$c(A) = c(D)$	[0, 1, 0, 0, -1, 0]
$c(B) + c(C) = c(D)$	[0, 0, -1, -1, 1, 0]
$c(B) + c(C) = c(E)$	[0, 0, -1, -1, 0, 1]
$c(D) = c(E)$	[0, 0, 0, 0, 1, -1]

Taulukko 3: Esimerkkikuvan 7 suorituspolusta muodostettuja yhtälöitä ja niistä johdettuja kertoimia.

Matriisista $\mathbf{X}$ voidaan johtaa kaksi aliavaruutta, $\mathbf{X}$:n riviavaruus ja $\mathbf{X}$:n nolla-avaruus, joka on ortogonaalinen komplementtiavaruus riviavaruudelle. Ortogonaalinen komplementtiavaruus on joukko vektoreita, jotka ovat kohtisuoria $\mathbf{X}$:n vektoreihin nähden. Nolla-avaruutta kutsutaan artikkelissa invarianttiavaruudeksi, ja vektorit invarianttiavaruudessa ovat invarianttivektoreita θ. Koska ortogonaalisten vektoreiden ominaisuus on, että niiden tulo on nolla, myös yhtälö 2 pitää paikkaansa.

Invariantti θ voi olla mikä tahansa vektori matriisin nolla-avaruudessa. Satunnaisen invarianttivektorin valitseminen invarianttiavaruudesta ei ole kuitenkaan kovinkaan mielekästä, sillä lyhyet jaksot ohjelman suorituksessa ovat yleensä mielekkäämpiä tarkastella kuin täysi suorituspolku koko järjestelmän suorituksen aikana. Esimerkkinä lyhyestä jaksosta on järjestelmän suorituspolun haarautumiset ja toistot. Esimerkkikuvassa 7 esiintyy haarautuma A:sta B:hen ja C:hen ja sitten yhdistyminen B:stä ja C:stä D:hen, ja näiden tarkastelu on mielekästä.

Perustavanlaatuiset suorituspolkut ovat yleensä hyvin yksinkertaisia, joten niiden invarianttivektorit ovat harvoja. Tämä voidaan kääntää myös niin, että harvat invarianttivektorit merkitsevät perustavanlaatuista suorituspolkua. Ylärajaa nollasta poikkeaville kertoimille merkitään $K(\mathbf{X})$, ja $K(\mathbf{X}) = m + 1 - r$, jossa m on viestityyppien määrä ja r on invarianttiavaruuden ulottuvuuksien määrä. Yläraja on siis sama kuin riviavaruuden ulottuvuuksien määrä. Kompaktiksi invarianttivektorijoukoksi kutsutaan joukkoa, joka ei sisällä sellaista invarianttivektoria, joka voitaisiin saada lineaarisena yhdistelmänä muista invarianttivektoreista samassa joukossa. Kompaktin invarianttivektorijoukon vastakohta on redundantti invarianttivektorijoukko. Esimerkiksi joukko $\{c(A) = c(D), c(A) = c(E), c(D) = c(E)\}$ on redundantti, koska $c(D) = c(E)$ voidaan päätellä kahden aiemmin invariantin avulla. Joukko $\{c(A) = c(D), c(A) = c(E), c(A) = c(B) + c(C)\}$ on sen sijaan kompakti joukko. Invarianttien määrä kompaktissa joukossa C ei voi olla suurempi kuin invarianttiavaruuden ulottuvuus, eli $|C| \leq r$. Invarianttivektorien ominaisuus on myös se, että niiden kertoimet voivat olla vain kokonaislukuja. Tässä menetelmässä pyritään löytämään järjestelmän suurin harva, kompakti ja vain kokonaislukuja sisältävä invarianttivektorien joukko.

Invarianttien etsiminen on kolmikohtainen menetelmä. Ensin jäsennetyt lokiviestipohjat ryhmitellään, ryhmien perusteella muodostetaan invarianttiavaruus ja lopuksi invarianttiavaruudesta löydetään merkitykselliset invariantit. Ryhmittelyn kannalta on hyvä ymmärtää vielä artikkelissa esiintyvä käsite parametri. Parametri on viestin osa, jonka arvo on jokin muuttuja. Sama muuttuja voi esiintyä parametrina useissa lokiviesityypeissä. Jos useampi parametri saa saman muuttuja-arvon, ne määritellään yhteisgeneettiseksi (cogenetic). Artikkelin kirjoittajat kertoivat tehneensä seuraavat havainnot:

- Kaksi parametriä P_a ja P_b ovat yhteisgeneettisiä jos niiden arvojoukko on tarkasteltavalla ajanhetkellä sama tai toisen arvojoukko on toisen alijoukko. Toisin sanoen $V_r(P_a) \subseteq V_r(P_b)$. Parametrin arvojoukko $V_r(P_a)$ on artikkelissa määritetty joukkona uniikkeja parametriarvoja lokijoukossa.

- Kaksi parametriä P_a ja P_b ovat suurella todennäköisyydellä yhteisgeneettisiä jos niiden arvojoukko ovat suurelta osin päällekkäisiä eli jos $V_r(P_a) \cap V_r(P_b)$ on suuri.

- Mitä pidempiä parametrien yhteisen arvojoukon arvot ovat, sitä suuremmalla todennäköisyydellä parametrit ovat yhteisgeneettisiä.

Artikkelin käyttämässä algoritmissa, jossa vertaillaan parametreja, on määritetty, että parametrit ovat yhteisgeneettisiä jos niiden päällekkäisten arvon osuus on suurempi kuin määritelty kynnysarvo ja parametrin arvon vähimmäispituus on 3. Näiden ehtojen mukaan muodostetaan parametriryhmiä, joiden sisällä parametrit ovat yhteisgeneettisiä, ja lokiviestit, jotka sisältävät yhteisgeneettisiä parametrejä ryhmitellään samoin yhteen. Osiossa 3.5 kerrotaan tarkemmin, miten jokaisesta ryhmästä muodostetaan joukko viestilukuvektoreita.

Ryhmittelyn jälkeen ryhmistä muodostetaan invarianttiavaruus ja etsitään invariantteja algoritmin 2 mukaisesti (Lou ja muut, 2010). Invarianttiavaruuden etsimisessä matriisista käytetään singulaariarvohajotelmaa (singular value decomposition). Singulaariarvohajotelmalla tarkoitetaan matriisin faktorointia $A = U\Sigma V^T$, joka jakaa matriisin singulaariarvoihin ja -vektoreihin. Singulaariarvot löytyvät diagonaalimatriisista Σ ja singulaarivektorit matriiseista U ja V. U:ssa on vasemmanpuoleiset ortogonaaliset vektorit ja V:ssä oikeanpuoleiset ortogonaaliset singulaarivektorit. Singulaariarvohajotelman avulla muodostetut oikeanpuoleiset singulaarivektorit (right-singular vector) kuvaavat potentiaalista invarianttiavaruutta. Vektorit käydään yksi kerrallaan läpi, ja yksittäinen vektori v_i on validoitu invariantti, jos viestiryhmistä 98%:lle pätee, että $|X_j v_i| < \epsilon$. Tässä X_j merkitsee viestilukuvektoria lokiviestiryhmässä j. Artikkelissa käytettiin kynnysarvona $\epsilon = 0{,}5$. Vektorit käydään läpi singulaariarvon mukaan aloittaen pienimmästä, kunnes ensimmäinen epävalidoitu invariantti löydetään. Kaikki siihen asti löytyneet validoidut invariantit muodostavat invarianttiavaruuden.

Algoritmi 2 Invarianttien louhiminen

1. Jokaiselle parametriryhmään liittyvälle viestiryhmälle rakennetaan matriisi X käyttäen viestimäärävektoreita ja estimoidaan invarianttiavaruuden ulottuvuus.
2. Raaka voima-algoritmia käyttäen etsitään sellaisia invariantteja, jotka sisältävät k nollasta poikkeavaa kerrointa, missä k:n arvo kasvaa yhdestä viiteen. Algoritmin suoritus lopetetaan kun jompi kumpi seuraavista ehdoista täyttyy:
   a. r riippumatonta invarianttia on löydetty;
   b. $k > (m - r + 1)$.
3. Jos $(m - r + 1) > 5$ ja ylempänä esitetyt ehdot eivät täyty, käytetään ahnetta algoritmia potentiaalisten invarianttien löytämiseksi kun $k > 5$.

Invariantteja etsitään raaka voima -algoritmilla, jossa nollasta eroavien kertoimien määrä k kasvaa joka iteraatiolla. Sen arvon on hyvä olla pieni, sillä suurella k:n arvolla etsintäavaruus kasvaa ja menetelmän tehokkuus huononee. Koska lokiviestityyppien välisistä suhteista ei todellisuudessa ole juurikaan tietoa, yritetään kaikkia hypoteettisia nollasta poikkeavia kertoimia eri ulottuvuuksissa potentiaalisen harvan invariantin rakentamiseksi ja sen jälkeen tarkistetaan, sopiiko se historialliseen lokidataan. Invarianttihypoteesi määritellään sen nollasta poikkeavana kerroinhahmona $\{p_j, j = 1, 2, \ldots, k\}$, jossa p_j on invarianttihypoteesin nollasta poikkeavan kertoimen indeksi, $0 \leq p_j \leq p_{j+1} \leq m$ ja k on nollasta poikkeavien kerrointen määrä.

Jokaiselle nollasta poikkeavalle kerroinhahmolle tarkastetaan, onko olemassa invarianttia $a_{p_j}, j = 1, 2, \ldots, k$. Tämä koostuu kahdesta osasta. Matriisista $\mathbf{X}$ valitaan k kolumnivektoria, joiden kolumni-indeksit ovat $\{p_j, j = 1, 2, \ldots, k\}$ ja rakennetaan niistä $\widehat{X}$. Sen avulla määritetään invarianttikandidaatti $\hat{\theta}$, joka noudattaa nollasta poikkeava kerroinhahmoa ja minimoi $\|\widehat{X}\hat{\theta}\|_0$. Toisin sanoen $\hat{\theta} = \text{argmin}_\theta(\|\widehat{X}\theta\|_0)$. Koska nollatermin optimointioperaatio on NP-kova ongelma, $\hat{\theta}$ estimoidaan $\|\widehat{X}\theta\|_2$:n avulla. Estimoidut θ:n kertoimet ovat yleensä välillä (-1,0, 1,0). Kuten aiemmin mainittiin, invarianttien tulee olla kokonaislukuja. θ skaalataan niin, että sen pienin nollasta poikkeava arvo on kokonaisluku l, $l = 1, 2, \ldots, p$ ja muutkin nollasta poikkeavat arvot pyöristetään kokonaislukuun. Tästä saadaan lopputuloksena p invarianttikandidaattia ja lopulliset invariantit valitaan tarkastamalla invarianttikandidaattien tukisuhde lokiviestiryhmissä. Jos tukisuhde ylittää käyttäjän määrittelemän kynnysarvon γ, invariantti määritetään validiksi. Algoritmin suoritus loppuu kun on löydetty r invarianttia tai k:n arvo kasvaa suuremmaksi kuin $(m - r + 1)$. Jälkimmäisessä tapauksessa käytetään artikkelissa (Donoho, Tsaig, Drori, & Starck, 2012) esiteltyä ahnetta menetelmää potentiaalisten invarienttien etsimiseksi. Ahne menetelmä ei kuitenkaan takaa kaikkien invarianttien löytymistä.

Anomalioiden havaitseminen uusista tapahtumista suoritetaan jäsentämällä ensin uudet lokiviestit, ryhmittelemällä ne ja muodostamalla viestimäärävektorit. Viestimäärävektoreita verrataan niihin liittyviin opittuihin invariantteihin, ja lokiviestiryhmä, jonka viestimäärävektori rikkoo siihen liittyviä invariantteja merkitään anomaliaksi. Löydettyyn anomaliaan liitetään invariantti, jota se rikkoo, niin että virhe on helpompi ymmärtää. Esimerkiksi jos virheeseen liittyy virhe c('Opening file ##')=c('Closing file ##'), on helppo nähdä että ongelma liittyy tiedoston avaamiseen ja sulkemiseen.

5 Menetelmien suoriutuminen

Aiemmassa luvussa mainittujen menetelmien suoriutumista on testattu yleensä joko injektoimalla virhetapauksia, jolloin tutkitaan havaitseeko menetelmä ne, tai käymällä lokitiedostoja manuaalisesti läpi, jolloin luokittelutietoa on käytetty vain validoimiseen. Manuaalisesti luokittelu on työlästä mutta mahdollistaa yleisten suoriutumismetriikoiden käytön. Yleisiä metriikoita ovat tarkkuus (accuracy), täsmällisyys (precision), erottelukyky (recall) ja F-mitta (F-measure). Tarkkuus lasketaan

$$\frac{OP + ON}{\text{Datajoukko}} = \frac{\text{Oikein luokitellut}}{\text{Datajoukko}},$$

täsmällisyys

$$\frac{OP}{OP + VP} = \frac{\text{Oikein havaitut anomaliat}}{\text{Kaikki raportoidut anomaliat}},$$

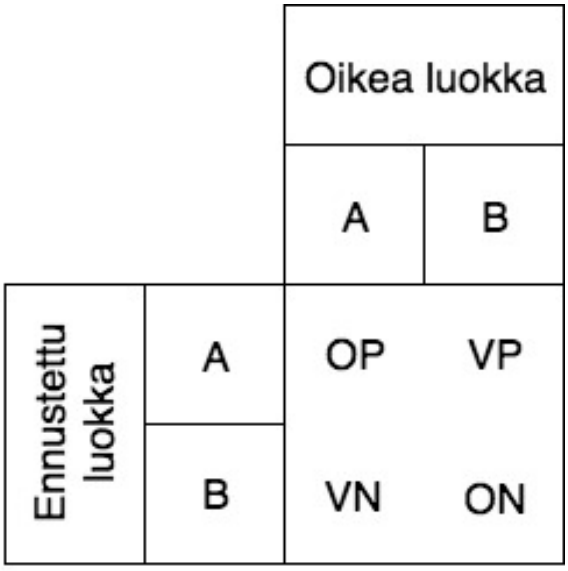
erottelukyky

$$\frac{OP}{OP + VN} = \frac{\text{Oikein havaitut anomaliat}}{\text{Kaikki anomaliat}},$$

ja F-mitta

$$\frac{2 \times \text{Täsmällisyys} \times \text{Erotteluykyky}}{\text{Täsmällisyys} + \text{Erotteluyky}}.$$

Nämä ovat yleisesti luokittelussa käytettyjä metriikoita (Stearley & Oliner, 2008; Chen ja muut, 2002). Näihin liittyvät käsitteet oikea positiivinen (OP), väärä positiivinen (VP), oikea negatiivinen (ON) ja väärä negatiivinen (VN) on esitetty virhematriisissa kuvassa 8.



Kuva 8: Virhematriisi.

5.1 Ryvästäminen

LogCluster-menetelmän suoriutumista mittaavassa tutkimuksessa (Lin ja muut, 2016) lokitiedostot jäsennettiin ensin vakioiksi ja muuttujiksi ja muodostettiin sitten lokisarjoja tapahtumatunnisteen mukaan. Lokisarjoista muodostettiin vektoreita ja ne painotettiin osiossa 4.1 esitettyjen menetelmien mukaan. Ryväksistä etsittiin edustavat vektorit ja ne käytiin läpi, jos ne eivät ole aiemmin esiintyneet.

LogCluster-menetelmän suoriutumista on tutkittu kahdessa Microsoftin tuotantojärjestelmässä ja kahdessa Hadoopia käyttävässä sovelluksessa. Tuotantojärjestelmien nimiä ei paljasteta artikkelissa, vaan niitä kutsutaan nimellä Palvelu X ja Palvelu Y ja Hadoopia käyttävät sovellukset ovat WordCount ja PageRank (Lin ja muut, 2016).

WordCount on MapReduce-ohjelmoinnin esimerkkinä luotu sovellus, jonka avulla voidaan laskea eri sanojen määrän annetuissa tiedostoissa. PageRank on sovellus, jota Google hyödyntää nettisivujen arvottamisessa. Molemmat käyttävät Hadoopia ja tässä tutkimuksessa käsitellään Hadoopin automaattisesti luomia lokiviestejä, ei sovellusten omia.

Tutkimuksessa tarkasteltiin sitä, kuinka paljon LogCluster vähentää työmäärää lokien tutkimisessa, kuinka tarkasti se tunnistaa anomaliat ja miten etäisyyden kynnysarvo vaikuttaa suoriutumiseen. Suoriutumista mitataan täsmällisyydellä ja normalisoidun jaetun informaation avulla. Tutkimuksessa vertailtiin menetelmän suoriutumista WordCountin ja PageRankin osalta avainsanojen etsimismenetelmään ja ICSE'13-menetelmään. ICSE'13-menetelmässä (Shang ja muut, 2013) muodostetaan suorituspolkuja kahden ympäristön tuottamista lokitiedostoista. Ympäristöt ovat pieni pseudoympäristö ja suuri pilviympäristö. Sen jälkeen näissä ympäristöissä muodostettuja suorituspolkuja vertaillaan toisiinsa ja raportoidaan eroavaisuudet. Eroavaisuuksista suodatetaan tiedetyt, alustasta johtuvat eroavaisuudet ja jäljelle jäävät eroavaisuudet tulkitaan anomalioiksi. Tutkimuksessa käytettäviin sovelluksiin WordCount ja PageRank injektoitiin kolmen tyyppisiä virhetapauksia: solmun kaatuminen, verkkoyhteysongelma ja levytilan täyttyminen. Menetelmien suoriutumista verrattiin, kun yksi ongelma esiintyi useasti peräkkäin tai kun kolme eri virhettä tapahtui. Tutkimuksen tuloksissa todetaan LogClusterin olevan huomattavasti tehokkaampi kuin vertailukohteet.

Taulukossa 4 esitellään tuloksia kolmen eri virheen sattuessa. 'Lokit'-sarake kertoo, kuinka monta lokisarjaa, tai avainsanamenetelmän tapauksessa raakaa lokiviestiä, joudutaan läpikäymään manuaalisesti lopuksi. 'Täsmällisyys'-sarake kertoo, kuinka monta prosenttia tutkituista lokijaksoista sisälsivät oikeita virheitä. Keskitarkkuus oli LogClusterille 60%, kun avainsanamenetelmälle se oli noin 20% ja ICSE'13 menetelmälle noin 30%.

Solmun kaatuminen						
	Avainsanat		ICSE'13		LogCluster	
	Lokit	Täsmällisyys	Lokit	Täsmällisyys	Lokit	Täsmällisyys
WordCount	335	16,7%	29	44,8%	8	100%
PageRank	361	1,1%	18	5,6%	2	50%
Verkkoyhteysongelma						
	Avainsanat		ICSE'13		LogCluster	
	Lokit	Täsmällisyys	Lokit	Täsmällisyys	Lokit	Täsmällisyys
WordCount	8437	97%	20	55%	6	66,7%
PageRank	10 250	14%	23	17,4%	2	50%
Levytilan täyttyminen						
	Avainsanat		ICSE'13		LogCluster	
	Lokit	Täsmällisyys	Lokit	Täsmällisyys	Lokit	Täsmällisyys
WordCount	388	24,5%	26	50%	6	100%
PageRank	395	0,8%	24	4,2%	4	25%

Taulukko 4: Menetelmien täsmällisyys ja läpikäytävien lokiviestien määrä, kun eri tyyppisiä virheitä (Lin ja muut, 2016).

Palvelu X on verkkopalvelu, jota käyttävät miljoonat ihmiset ympäri maailmaa. X on usealla palvelimella ajossa, ja käytössä on kuormanjakaja. Käyttäjän tekemät pyynnöt kulkevat usean komponentin ja palvelimen läpi, ja jokainen niistä luo omat lokitiedostonsa. Lokitiedostot yhdistetään automaattisesti HDFS:n kaltaiseen datavarastoon. Kokeessa käytettävässä lokitiedostossa on 3,3 miljoonaa lokiviestiä. Palvelu Y on myös laaja palvelu, jonka periaatteena on toimia ympärivuorokautisesti ja olla 99,99% saavutettavissa. Palvelu sisältää monta alijärjestelmää, joista jokainen on jaettu 3-kerrosarkkitehtuurin mukaisesti käyttöliittymään, sovelluskerrokseen ja tietokantaan. Artikkelin tutkimuksessa palvelusta Y on mukana 10 miljoonaa lokiviestiä. Taulukossa 5 esitellään tuloksia näille palveluille. Niiden suoriutumisen tutkiminen eroaa WordCountista ja PageRankista siten, että niihin ei injektoitu virheitä. Ryvästen etäisyyskynnysarvon θ ei todettu vaikuttavan juurikaan tuloksiin, kun se oli välillä 0,2 – 0,8. Ryvästen laatua on myös tarkasteltu normalisoidun jaetun informaation (normalized mutual information) avulla. Se on kaikkien järjestelmien osalta yli 80% , kun kynnysarvo θ=0,5, joten ryvästäminen tuottaa hyvälaatuisia ryväksiä.

	Avainsanat		**ICSE'13**		**LogCluster**	
Palvelu	Lokeja	Täsmällisyys	Lokeja	Täsmällisyys	Lokeja	Täsmällisyys
X	278 430	0,01%	552	0,77%	7	42,86%
Palvelu	Lokeja	Täsmällisyys	Lokeja	Täsmällisyys	Lokeja	Täsmällisyys
Y	200 119	0,08%	2433	2,84%	40	55%

Taulukko 5: Eri menetelmien täsmällisyys ja läpikäytävien lokiviestien määrä Microsoftin palveluissa (Lin ja muut, 2016).

Myös Pinpointia arvioitiin injektoimalla virhetilanteita kokeellisessa ympäristössä (Chen ja muut, 2002). Pinpointin suoriutumista arvioitiin käyttämällä J2EE PetStore -sovellusta, ja virheet saattoivat koskea yhtä tai useampaa sovelluksen komponenttia. Injektoituja virheitä olivat esitellyt (declared) poikkeukset, suorituksen aikaiset poikkeukset, ikuiset silmukat ja null-kutsut. Ne kattavat hyvin erilaiset virhetilanteet: mukana on ennustettuja ja ennalta-arvaamattomia poikkeuksia ja virheitä, jotka eivät välttämättä aiheuta lainkaan poikkeusta, mutta joko jumittavat järjestelmänn tai saattavat aiheuttaa järjestelmän väärin toimimisen. Virheen injektoiva järjestelmä on erillään virheitä havaitsevasta järjestelmästä.

Tutkimusta varten rakennettiin selainemulaattori, joka tallentaa käyttäjän toimet selaimessa ja toisintaa ne sitten useasti sovelluksen testaamiseksi. Sovelluksessa esitettyjä toimintoja olivat haku, yksittäisen asian tarkastelu, uuden tunnuksen luonti, käyttäjäasetusten vaihtaminen, tilausten lisääminen ja tilausten loppuunvieminen. Sovelluspalvelin käynnistettiin uudelleen virheiden välissä, jotta ne eivät vaikuttaisi toisiinsa. Testejä ajettiin 133 kertaa viisi minuuttia kerrallaan niin, että yksi transaktio oli aktiivinen kerrallaan.

Pinpointin tarkkuutta verrataan artikkelissa kahteen muuhun virheenhavaitsemismenetelmään. Ensimmäinen on samankaltainen kuin monitorointijärjestelmä: se yksinkertaisesti palauttaa komponentin, jossa virhe esiintyi. Menetelmää kutsutaan ar-

tikkelissa nimellä 'havaitseminen'. Toinen menetelmä palauttaa kaikki komponentit, joissa epäonnistunut pyyntö kulki. Tässä menetelmässä voidaan asettaa jokin kynnysarvo sille, monessako prosentissa epäonnistuneita pyyntöjä komponentin täytyy olla mukana, jotta se merkitään potentiaaliseksi virheeksi. Menetelmää kutsutaan 'riippuvaisuudeksi'.

Pinpoint suoriutuu hyvin verrattuna kahteen muuhun menetelmään etenkin siinä tapauksessa, että virhe sattuu vain yhdessä komponentissa. Siinä tapauksessa Pinpoint saavuttaa jopa 80-90% tarkkuuden kun täsmällisyys on 50-60%. Havaitsemismenetelmän tarkkuus pysyy noin 50 prosentissa ja väärien positiivisten osuus on pahimmillaan saman verran, kun riippuvuusmenetelmän tarkkuus on melko korkea, mutta väärien positiivisten osuus on myös hyvin korkea.

Epäilypisteeseen perustuvan ryvästysmenetelmän suoriutumista tutkittiin hajautetussa ryväksen hallintajärjestelmä SCoressa (Mirgorodskiy ja muut, 2006). Tutkimus suoritettiin laajassa tuotantoympäristössä. Tutkimuksessa on jaettu mahdolliset ongelmat neljään kategoriaan. Ensimmäinen niistä on epädeterministinen virhelopetus (non-deterministic fail-stop) ongelma, joka tapahtuu, kun prosessi vikaantuu ja sen suorituspolku loppuu ennenaikaisesti. Toinen virhekategoria on ikuiset silmukat, joka tarkoittaa sitä, että prosessi ei kaadu, mutta jää suorittamaan tiettyä osaa koodista. Kolmanteen kategoriaan kuuluu lukkiutumis-, elolukko- ja nälkiintymisongelmat. Lukkiutumisella tarkoitetaan tilannetta, jossa prosessit jäävät odottamaan muita prosesseja ristiin, eikä mikään niistä pääse eteenpäin. Se näkyy suorituspolussa siten, että lukkiutuneet prosessit lopettavat lokien tuottamisen ja niiden suorituspolut koodissa eroavat muista prosesseista. Elolukko on samankaltainen tilanne, paitsi että jokaisen prosessin tila vaihtuu jatkuvasti ja mikään niistä ei pääse etenemään. Nälkiintyminen tapahtuu, kun muut prosessit priorisoidaan jatkuvasti jonkin prosessin edelle niin, että se ei saa koskaan vuoroaan suorittaa. Viimeinen kategoria on kuorman epätasainen jakautuminen. Se on yksi pääongelmista rinnakkaisissa järjestelmissä ja esiintyy yksittäisen solmun vähäisenä työmääränä. Tässä tutkielmassa keskitytään näistä kaikkiin paitsi ensimmäiseen, sillä ensimmäisen syy on helppo löytää suorituspolusta ilman koneoppimista.

SCore on laaja rinnakkainen ohjelmointiympäristö työasemien ryväksille. Käyttäjät voivat toimittaa tehtävän SCoren keskitetylle ajastimelle, joka määrittää, montako laskentasolmua tehtävälle varataan, ja ajastaa sitten tehtävän suorituksen. Kun tehtävää suoritetaan, sen tilaa valvotaan solmuilla ja lopuksi, kun tehtävä on valmis, sille varatut solmut vapautetaan. SCoreen kuuluu esimerkiksi hajautettujen tehtävien ajastus, rinnakkaisten prosessien migratointi ja jaetun muistin infrastruktuuri. Sillä on oma menetelmä, jolla se valvoo laitteisto- ja ohjelmistovikoja, ja kun vika havaitaan, se pyrkii lopettamaan kaikki käynnissä olevat prosessit ja käynnistämään ne uudelleen. SCoren lähdekoodi koostuu yli 200 000 rivistä C++-koodia yli 700:ssa tiedostossa.

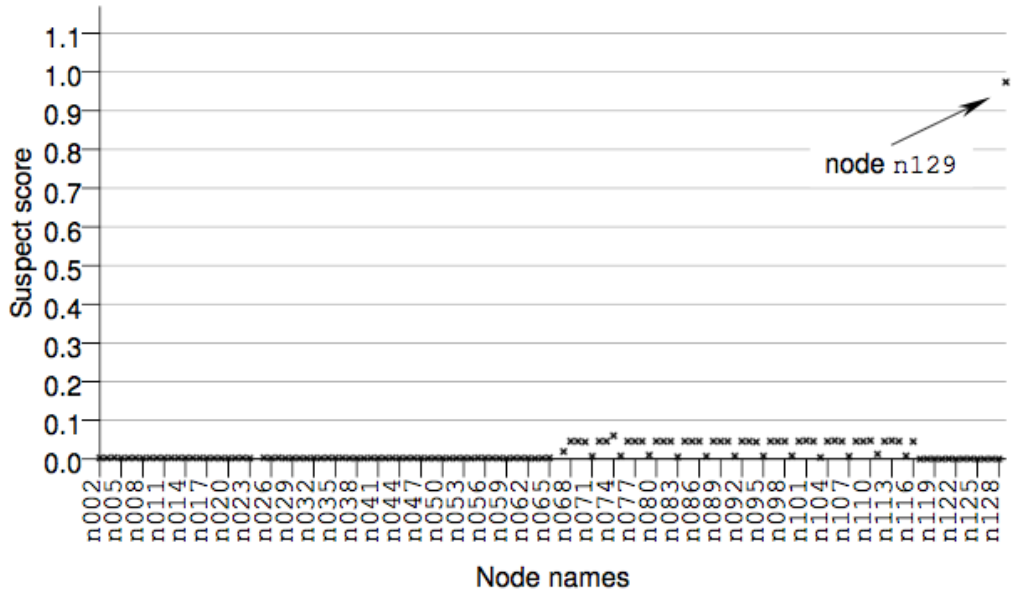
SCorea seurattiin kolmen kuukauden ajan 129-solmuisessa laskentaryväksessä. Tänä aikana havaittiin useita virhetilanteita ja toimija spTracer injektoitiin jokaiselle ryväksen solmulle ja kerättiin lokitietoja funktiotasolla. Kun SCoren oma järjes-

telmä huomaa virheen ja prosessit lopetetaan, spTracer tallentaa lokitiedot ja ne analysoidaan virheen analysoimiseksi. Eräs virhe tapahtui SCoren komponentissa sbcast, joka on lähetyspalvelu valvojasolmulla. Se kokoaa scored-prosessien tuottamaa monitorointi-informaatiota mahdollistaen siten asiakasohjelmalle monitorointidatan noutamisen yhdestä paikasta sen sijaan, että se pitäisi hakea kaikilta solmuilta. Eräässä suorituksessa sbcast lopetti saapuviin pyyntöihin vastaamisen ja koko SCore-järjestelmä lopetti toimintansa. Tapahtumahetkellä ei ollut selvää, että virhe tapahtui sbcastissa.

Virhettä selvittäessä selvitettiin ensin, että kyse ei ole virhe-lopetus ongelmasta. Koska solmujen lopetuksen aikaleimojen välillä ei ollut suurta eroa, kyse ei ollut siitä. Kun tämä oli selvää, käytettiin osiossa 4.1 esiteltyä menetelmää epäilypisteiden laskemiseksi. Eräs solmu sai erityisen korkeat epäilypisteet ja nousi todennäköiseksi syylliseksi. Kuvassa 9 näkyy epäilypisteet solmuittain. X-akselilla näkyvät solmujen nimet ja y-akselilla niiden epäilypisteet. Sama virhe ei toistunut muissa solmuissa, mutta virhe saatiin jäljitettyä suorituspolkuun. Yksittäisten funktioprofiileiden sijaan voidaan siis myös muodostaa suorituspolkuprofiileja. Esimerkiksi polut $(A \rightarrow B \rightarrow C \rightarrow D)$ ja $(E \rightarrow F \rightarrow D)$ ovat erillisiä suorituspolkuja ja $t(h, f_1)$ on suorituspolun A:sta D:hen suoritusaika ja $t(h, f_2)$ on polun E:stä D:hen suoritusaika. A, B, C, D, E ja F ovat lähdekoodin funktioita. Tutkimuksessa suorituspolkuprofiileja hyödyntäen löydettiin, että suorituspolulla (output_job_status → score_write_short → score_write → __libc_write) oli suurin vaikutus epäilypisteisiin. Funktion score_write kutsusta ei koskaan palattu, koska se kutsui funktiota __libc_write silmukassa ja näin aiheutti sen, että SCoren sisäinen monitorointijärjestelmä lopetti käynnissä olevat prosessit ja käynnisti ne uudelleen. Ongelman tutkimista jatkettiin manuaalisesti lähdekoodista ja havaittiin, että virhetilanteen aiheutti scored-prosessi, joka yritti kirjoittaa lokiviestiä pistokkeeseen. Pistoke oli yhteydessä scbcastiin, jota spTracer ei valvonut. Tämän tyyppinen virhe sattuu jos scbcast lopettaa suorituksen ja pistokkeen kuuntelemisen. Kyseisen menetelmän avulla löydettiin siis olemassa oleva ohjelmistovirhe.

5.2 Painotusmenetelmät

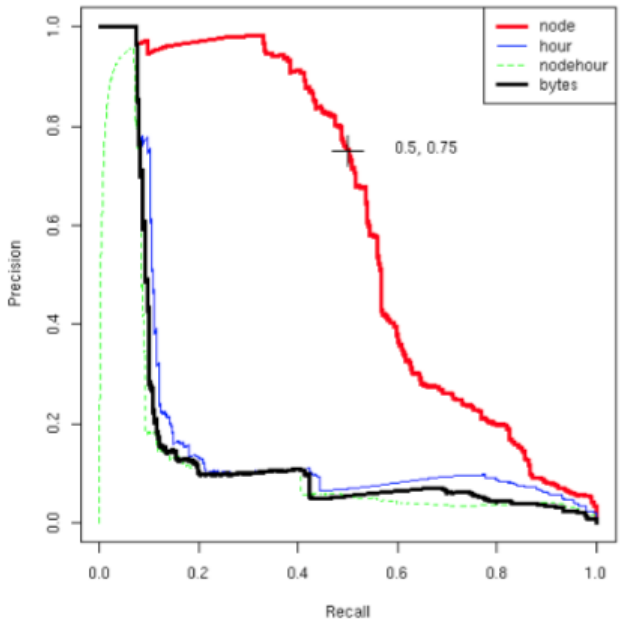
Painotusmenetelmien suoriutumista tutkittiin 512:n solmun ryväksessä, Spiritissä (Stearley & Oliner, 2008). Lokeja kerättiin 23 päivän ajalta ja lokiviestejä oli yhteensä 8,3 miljoonaa kappaletta, eli yhteensä 911 megatavua. Menetelmän tuottaman suuruusluokkaan (magnitude) perustuvan luokittelijan tarkkuus ei ole kovinkaan korkea, paitsi osiossa 4.2 esiteltyjen aggregaattimatriisien $\mathbf{Y}$ ja $\mathbf{Z}$ osalta ja vain log.entropy -painotuksen tuottaman luokittelijan osalta. Matriisin $\mathbf{Y}$ suuruusluokkamenetelmää, joka on tuotettu log.entropylla, kutsutaan solmuluokittelijaksi ja matriisin $\mathbf{Z}$ suuruusluokkamenetelmää tuntiluokittelijaksi. Alkuperäiseen matriisin $\mathbf{X}$ suuruusluokkamenetelmää, joka on saatu log.entropya käyttämällä kutsutaan solmutuntiluokittelijaksi.



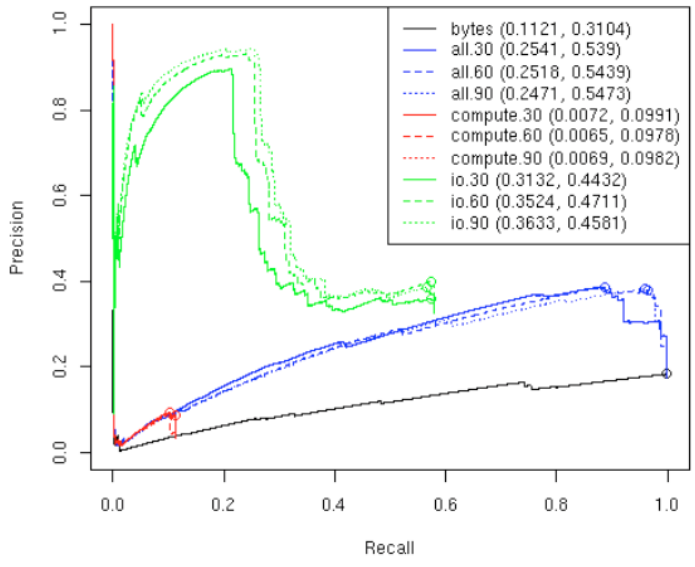
Kuva 9: Epäilypisteytys SCoressa (Mirgorodskiy ja muut, 2006).

Kuvassa 10 esitetään tulokset. Täsmällisyys - erottelukyky käyrä muodostetaan vaihtelemalla täsmällisyyden kynnysarvoa suurimmasta pienimpään. Vertailukohdaksi on otettu hyvin yksinkertainen luokittelija, joka tunnistaa anomaliat tavujen määrästä per solmutunti. Menetelmää kutsutaan tavuluokittelijaksi. Jos luokittelija ei yllä sitä parempiin tuloksiin, se ei suoriudu kovinkaan hyvin eikä ole kiinnostava. Tavuluokittelija havaitsee lähinnä purskeiset virhetilanteet, jotka tutkijoiden mukaan muodostavat noin 10% virhetilanteista. Kuten kuvasta huomataan, tunti- ja solmutuntiluokittelijat eivät pärjää paremmin kuin tavuluokittelija, mutta solmuluokittelija tuottaa hyviäkin tuloksia. Virheen sisältävistä solmutunneista 50% havaitaan ennen kuin täsmällisyys putoaa alle 75%:iin.

Tutkimuksessa parhaiten menestyneen solmuluokittelija Nodeinfon suoriutumista mitattiin neljässä käynnissä olevassa ryväksessä: Blue Gene/L:ssä, Thunderbirdissä, Spiritissä ja Libertyssä (Oliner ja muut, 2008). Vertailukohtana oli tässäkin tapauksessa tavuluokittelija. Nodeinfoa muokattiin toimimaan liukuvissa ikkunoissa ja sitä ajettiin erikseen toiminnallisissa ryhmissä. Tutkimuksessa haettiin aina $W - 1$ päivän tiedot nykyisten solmutuntien laskemiseen. Tutkimuksessa testattiin menetelmää kun W on 30, 60 ja 90 päivää. Esimerkiksi kun W=30 ja solmutunnit halutaan generoida toukokuun 30. päivälle, käytetään siihen dataa, joka on generoitu toukokuun 1. klo 00:00 jälkeen. Tulokset BlueGene/L:n osalta näkyvät kuvassa 11. Kuvassa on näkyvissä tulokset kun Nodeinfoa on käytetty kaikilla solmuille (all) tai jaoteltu solmujen toiminnallisuuden mukaan (compute, io). Perässä näkyvät 30, 60 ja 90 viittaavat W:n arvoon, ja suluissa olevat luvut alueeseen käyrän alla (area under curve) ja suurimpaan F-mittaan. Käyttäen suoriutumiskriteerinä aluetta käyrän



Kuva 10: Tutkimustulokset esitellylle log.entropy -menetelmälle kun on käytetty aggregoitua tai aggregoimatonta dataa. Bytes viittaa tavuluokittelijaan. Node on solmuluokittelija, hour on tuntiluokittelija ja nodehour solmutuntiluokittelija (Oliner ja muut, 2008).



Kuva 11: Nodeinfo-menetelmän suoriutuminen käynnissä olevassa Blue Gene/L -ryväksessä (Oliner ja muut, 2008).

alla parhaiten suoriutui io-solmuissa käytetty Nodeinfo kun W:n arvo on 90, eikä tulos ei ole muillakaan W:n arvoilla paljon huonompi. Aluetta käyrän alla käytetään binääriluokittelussa suoriutumisen mittana kun positiivisten määrä datassa on huomattavasti pienempi kuin negatiivisten määrä, ja luokittelijasta mitataan täsmällisyyttä ja erottelukykyä. Mitä suurempi alue käyrän alla on, sitä paremmin luokittelija suoriutuu.

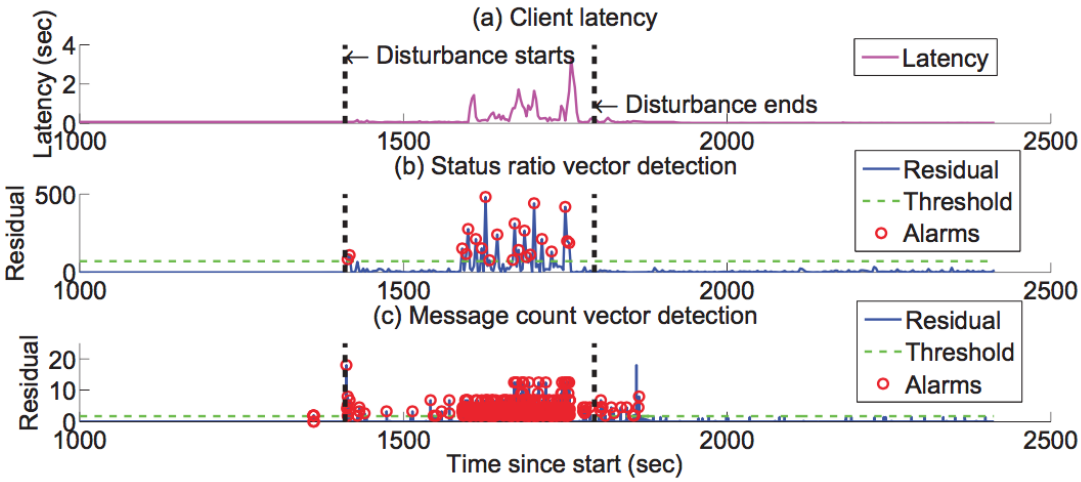
5.3 Pääkomponenttianalyysi

Pääkomponenttianalyysin suoriutumista on tutkittu kolmella eri tutkimusasetelmalla. Ensimmäinen tutkimus on suoritettu käyttäen koko olemassa olevaa lokidataa (Xu ja muut, 2009a), toinen käyttämällä samaa dataa mutta ottamalla siitä satunnaisotannalle pienempiä otoksia tuotantotilanteen simuloimiseksi (Xu ja muut, 2009b), ja kolmas tutkimus on suoritettu aidossa tuotantojärjestelmässä (Xu, Huang, & Jordan, 2010). Lokiviestit siis ensin jäsennetään lähdekoodista osiossa 3.3 kuvatulla tavalla ja niistä muodostetaan vektorit ja edelleen matriisit osiossa 3.5 kuvatulla tavalla. Matriiseihin sovelletaan sitten pääkomponenttianalyysiä kuten osiossa 4.3 on kuvattu.

Pääkomponenttianalyysin suoriutumista mitattiin kahdessa eri järjestelmässä, DarkStarissa ja HDFS:ssä (Xu ja muut, 2009a). DarkStar on verkkopelipalvelin, joka keskittyy lähinnä pieniin, aikaherkkiin transaktioihin. Se sijoitettiin yhdelle solmulle ja lokitusastetta nostettiin sallimalla myös 'DEBUG'-lokitustason lokitus. DarkStarille laitettiin DarkMud-niminen peli ja emulaattori, joka emuloi 60 käyttäjän toimintaa pelissä 4800 sekunnin ajan. Koska DarkStarin oli huomattu olevan herkkä kuormituksen vaihtelulle, prosessin käyttöä rajoitettiin 50% normaalista 1400 ja 1800 sekunnin välillä. HDFS puolestaan on suuria tiedostoja ja eräkäsittelyä varten suunniteltu tiedostojärjestelmä, joka DarkStariin verrattuna on suuremman skaalan järjestelmä ja paljon monimutkaisempi. HDFS:stä kerättiin yli 24 miljoonaa lokiviestiä. Datan koko oli 2,4 gigatavua, ja se sisälsi 575 319 tapahtumaketjua, joista uniikkeja oli vain 680.

DarkStarin lokeista eristettiin kahdeksan tilaa, joten tilasuhdematriisissa oli kahdeksan ulottuvuutta. Automaattisesti määritetty ikkunan koko oli kolme sekuntia. Emulaattori tallentaa käytössä esiintyvän viiveen, joten sitä verrataan havaittuihin anomalioihin. Tuloksista huomattiin, että menetelmän löytämät anomaliat täsmäsivät hyvin emulaattorin tallentamiin kohtiin, joissa viive kasvoi. ABORTING-tilan suhde COMMITTING-tilaan kasvoi 1:2000:sta jopa 1:2:een, mikä olisi vaikeaa huomata käyttämällä grepiä ja hakemalla avainsanoja. Menetelmän suoriutumista HDFS:ssä ei kerrottu juurikaan, koska ainoa tila, joka tunnistettiin, oli solmun nimi. Koska tutkimuksessa oli mukana 203 solmua ja niiden nimet täyttivät menetelmän tilan määritelmän, solmun nimi oli täysin käypä tila. Tilasuhdevektori onkin tässä tapauksessa oikeastaan vektori, joka mittaa aktiivisuutta per solmu, ja koska kyseistä piirrettä on tutkittu jo paljon, pääkomponenttianalyysimenetelmä ei tuo siihen juuri uutta (Xu ja muut, 2009a).

Myös viestimäärävektoreiden suoriutumista mitattiin molemmissa järjestelmissä. Algoritmi valitsi DarkStarista automaattisesti kaksi käytettävää tunnistetta, jotka ovat transaktiotunniste ja asynkroninen kanavatunniste (asynchronous channel id). Lokeissa esiintyi 68 029 transaktiotunnistetta yhteensä 18:ssa eri viestityypissä, joten viestimäärämatriisin ulottuvuus oli $68\,029 \times 18$. Pääkomponenttianalyysi tunnisti yleisiä tapahtumia vastaavat vektorit. Yleisten tapahtumien joukko oli {create, join txn, commit, prepareAndCommit}. Epänormaalit transaktiot eroavat tästä esimerkiksi siten, että jokin viestityypeistä puuttuu tai jokin harvinainen viestityyppi esiintyy siinä. Epänormaaleja transaktioita löydettiin yhteensä 504 ja niiden ajankohta tunnistettiin valitsemalla viimeisen tapahtuman aikaleima. Viestimäärävektoreiden avulla tehtävä anomalian tunnistus merkitsi jopa häiriöiden jälkeen tapahtuvan latenssin, joka johtui järjestelmän jonomenetelmästä. Tilasuhdevektoreihin perustuva anomalioiden tunnistus ei havainnut tätä, koska se kokoaa toimintaa ja viestimäärävektorit taas mallintavat yksittäisiä operaatioita. Tarkemmin tulokset näkyvät kuvassa 12.



Kuva 12: DarkStarissa suoritetun tutkimuksen tulokset. Ylimmällä rivillä näkyy emulaattorin mittaama viive sekunteina, toisessa näkyy tilasuhdevektoreista havaitut anomaliat ja kolmannessa viestimäärävektoreista havaitut anomaliat. Pystyviivat merkitsevät häiriöiden alku- ja loppuajankohtaa.

HDFS:stä valittiin vain yksi tunniste, lohkotunniste, joita löytyi 575 139 uniikkia tapausta 29:ssä eri viestityypissä. Viestimäärämatriisin ulottuvuus oli siis $575\ 139 \times 29$. Käyttäen automaattista kynnysarvoa Q pääkomponenttianalyysi tunnisti epänormaalit vektorit, jotka vastasivat lohkoja, jotka esiintyivät epänormaaleissa suorituspoluissa. Tutkijat kävivät löydetyt anomaliat läpi manuaalisesti ja löysivät 11 eri syistä johtuvaa anomaliaa. Yhden anomalian kuvaus on 'Namenodea ei päivitetä lohkon poistamisen jälkeen', ja kyseessä on pitkään piilossa ollut vika HDFS:ssä. Vika liittyi erääseen harvinaiseen suorituspolkuun ja on hankala havaita, mutta löydettiin pääkomponenttianalyysin avulla, koska mikään yksittäinen virheviesti ei riitä osoittamaan ongelmaa. Mukana oli toki myös vääriä positiivisia, jotka jaettiin kol-

meen kategoriaan. Kyseessä oli harvinaisia, mutta kuitenkin normaaleja tapahtumia. Oikeista negatiivisista esimerkiksi nostettiin tapahtuma, jota pääkomponenttianalyysi ei merkinnyt anomaliaksi. Tapahtumaa '#:Got Exception while serving # to #:#' luullaan tyypillisesti virheeksi, mutta se kuvaa todellisuudessa HDFS:n normaalia toimintaa. Esimerkiksi avainsanalla haku olisi todennäköisesti palauttanut kyseisen tapahtuman. Pääkomponenttianalyysi löysi 16 916 anomaliasta 10 217, ja kun menetelmää tehostettiin käyttämällä käänteistä dokumenttitiheyttä, tulos parani 16 808:aan. Vastaavasti väärien positiivisten määrä väheni 1797:stä 1746:een.

Toisessa pääkomponenttianalyysin suoriutumista mittaavassa tutkimuksessa käytettiin samaa HDFS:n lokidataa, mutta niin, että mallin muodostamiseen käytettiin vain 10% satunnaisotannalla valittua lokisarjaa (Xu ja muut, 2009b). Tämä tehtiin, jotta tilanne vastaisi sitä, miten menetelmää oikeasti käytettäisiin tuotannossa. Satunnaisotannan perusteella etsittiin yleiset hahmot, hahmojen kestojakauma ja pääkomponenttianalyysia käyttävä anomalioiden tunnistaja. Tämän jälkeen kaikkia suorituspolkuja käytettiin ongelman havaitsemiseen käyttäen muodostettua mallia. Tutkimus suoritettiin useaan otteeseen eri otoksilla, ja tulokset olivat yhdenmukaiset, koska tunnistetut hahmot ovat niin yleisiä, että satunnaisotoksen ottaminen ei vaikuta menetelmän suoriutumiseen.

Koska anomalioita tunnistetaan uusista tapahtumista, täytyy määritellä parametri sille, kuinka paljon viivettä tunnistuksessa saa olla. Tätä merkitään T_{max} ja se asetettiin 60 sekuntiin, eli anomalia pitää tunnistaa viimeistään 60 sekuntia siitä, kun epäilty tapahtumapolku esiintyy lokeissa. Pääkomponenttianalyysin kynnysarvoksi α määritettiin 0,001. Tutkimuksessa saatiin ensimmäisessä vaiheessa, jossa etsitään toistuvia hahmoja, karsittua datasta 85,6% ja pääkomponenttianalyysi suoritettiin enää 14,4%:lle. Esimerkki löydetystä hahmosta on 'Allocate a block for writing' ja sen suorituksen keston jakaumassa 99,9, 99,5 ja 99,99 prosenttipisteet ovat 11, 13 ja 20 sekuntia eli hyvin paljon vähemmän kuin määritetty T_{max}. Prosenttipisteiden arvot ovat tärkeitä tehokkuuden kannalta, sillä havaitsemisviive perustuu aina joko hahmon kestoon tai T_{max}:n arvoon. Tutkimuksen tarkemmat tulokset näkyvät taulukoissa 6 ja 7 ja taulukossa 6 näkyvä α merkitsee luottamustasoa.

α	OP	VP	VN	Täsmällisyys	Erottelukyky
0,0001	16 916	2 444	0	87,38%	100%
0,001	16 916	2 748	0	86,03%	100%
0,005	16 916	2 914	0	85,31%	100%
0,01	16 916	2 914	0	85,31%	100%

Taulukko 6: Tulokset eri luottamustasolla, kun $T_{max} = 60$s (Xu ja muut, 2009b).

Havaitsemisviive pysyi matalana ja yli 80% tapahtumista saatiin luokiteltua muutamassa sekunnissa. Tämä johtuu siitä, että useimmille tapahtumille voidaan käyttää $T_{katkaisu}$ määrittämään, kuinka kauan odotetaan tulevia tapahtumia ja T_{max} käytetään käytännössä vain kun tapahtumasarjalle ei löydy vastaavaa pohjaa.

T_{max}	OP	VP	VN	Täsmällisyys	Erottelukyky
15	2 870	129	14 046	95,70%	16,97%
30	16 916	2 748	0	86,03%	100%
60	16 916	2 748	0	86,03%	100%
120	16 916	2 748	0	86,03%	100%
240	14 23	2 232	2 683	86,44%	84,14%

Taulukko 7: Tulokset eri T_{max} arvoilla, kun $\alpha = 0,001$ (Xu ja muut, 2009b).

Edellä esitetyn kahden tutkimuksen tuloksia verrattiin myös toisiinsa arvoilla $\alpha = 0.001$ ja $T_{max} = 60$. Oikeaa reaalimaailman tilannetta simuloiva pärjäsi paremmin kuin menetelmä, jossa dataa ei jaettu osiin, koska se löysi kaikki anomaliat eikä väärien positiivistenkaan määrä kasvanut kuin muutamalla kymmenellä. Tämä johtuu siitä, että viestisarjat jaetaan sessioihin ja anomalioiden havaitseminen tehdään sessioille eikä kokonaisille poluille. Näin ne sisältävät vähemmän kohinaa (Xu ja muut, 2009b). Tarkemmat tulokset on esitetty taulukossa 8, jossa toisen tutkimuksen tulokset ovat sarakkeessa 'Online' ja ensimmäisen tutkimuksen tulokset sarakkeessa 'Offline'.

Anomalian kuvaus	Todellinen määrä	Offline	Online
Niminodea ei päivitetty lohkon poistamisen jälkeen.	4297	4297	4297
Kirjoituspoikkeus, asiakassovellus luovutti.	3225	3225	3225
Kirjoitus epäonnistui aloituksessa.	2950	2950	2950
Kopio poistettiin heti.	2809	2778	2809
Vastaanotettiin lohko, joka ei kuulu mihinkään tiedostoon.	1240	1228	1240
Tarpeeton addStoredBlock.	953	953	953
Poistettiin lohko, jota ei ollut enää datasolmulla.	724	650	724
Tyhjä paketti lohkolle.	476	476	476
Vastaanotettiin lohkopoikkeus.	89	89	89
Kopiointimonitorin aikakatkaisu.	45	45	45
Muita anomalioita.	108	107	108

Taulukko 8: Pääkomponenttianalyysin tulosten vertailua käyttäen kaksivaiheista menetelmää ja ilman sitä (Xu ja muut, 2009b).

Pääkomponenttianalyysiin perustuvan menetelmän suoriutumista tutkittiin myös Googlen GX-nimisessä tuotantojärjestelmässä (Xu ja muut, 2010). GX on hajautettu varastointijärjestelmä (storage system), johon kuuluu tuhansia solmuja. GX:n tuottamat lokit ovat tavallisia tekstitiedostoja, ja tutkittava data koostuu lokiviesteistä, jotka on kerätty kahden kuukauden aikana.

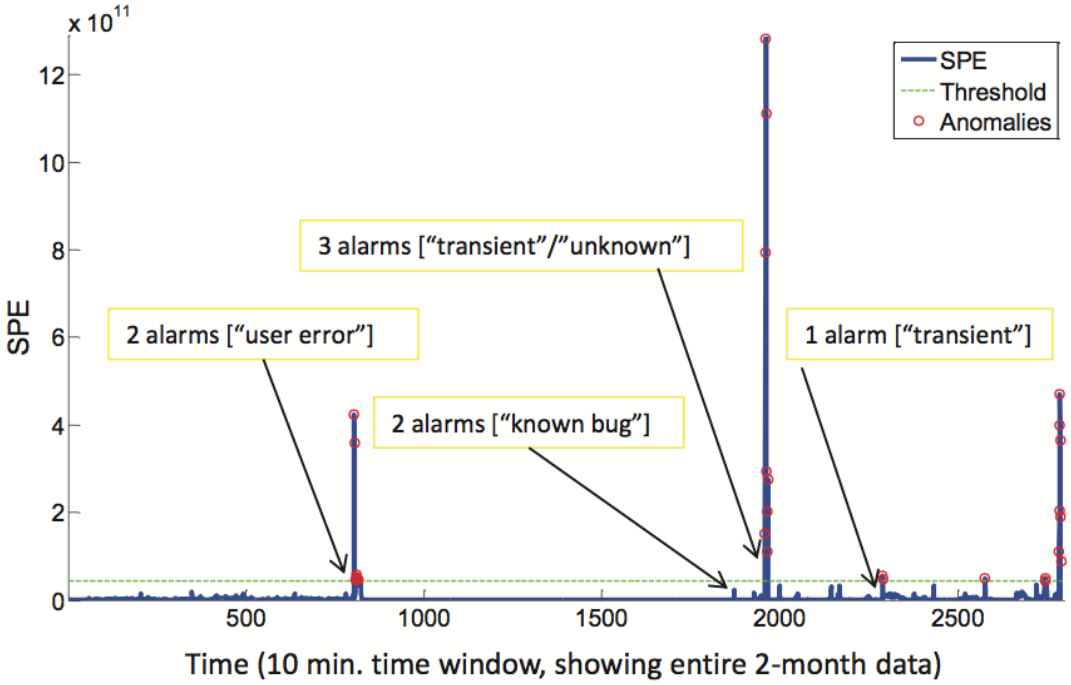
GX:n tuottamien lokien määrä on huomattavasti suurempi kuin kahdessa aiemmin mainitussa tutkimuksessa. Lokeihin kirjoitetaan vain virheet ja ajastetut taustatehtävät, mutta yhdeltä solmulta saattaa silti kerääntyä lokiviestejä yli puoli miljoonaa päivässä, ja koko järjestelmästä yli miljardi viestiä. Tutkimuksessa käytetty data sisältää kymmeniä miljardeja viestejä eli 400 gigatavua pakkaamatonta dataa. Tästä syystä menetelmä suoritetaan rinnakkaisesti. Samasta syystä myös tässä tutkimuksessa käytetään kaksivaiheista menetelmää. Tutkimuksen tuloksista ei ole tarkkaa metriikkaa, koska lokidata ei ole valmiiksi luokiteltua (Xu ja muut, 2010).

Lokiviestien malli eristetään ensin lähdekoodista osiossa 3.3 kuvatulla tavalla. GX on rakennettu C:llä ja C++:lla ja on riippuvainen monista muista projekteista ja kirjastoista, jotka luovat myös lokiviestejä, joten jos jokin lokiviesti ei vastaa louhittuja viestimalleja, sen vastaavuutta yritetään löytää niistä kokotekstihaulla. Projektien karsimiseksi käytetään heuristiikkaa useiten käytettyjen projektien löytämiseksi. Kun viestipohjat on luotu lähdekoodin pohjalta, niistä rakennetaan indeksi ja suorituksenaikainen lokien jäsentäminen on vain lähimmän viestipohjan etsimistä. Tämä prosessi on helppo hajauttaa, ja koko datan läpikäyminen kestää vain muutaman tunnin. Seuraavaksi suoritetaan piirrevalinta ja luodaan tilasuhdevektorit ja viestimäärävektorit. Tilasuhdevektorit osoittautuivat erityisen hyviksi anomalioiden tunnistamiseksi, kun viestien jäsentämisessä lisättyä viestityyppitunnistetta käytetään erityisenä tilamuuttujana.

Suurin osa GX:n lokitiedostoissa esiintyvistä lokeista on peräisin taustatehtävien suorituksista, joissa uudelleenjärjestellään dataa. Jokainen taustatehtävä luo aina saman joukon viestityyppejä ja eri taustatehtävien tuottamat viestityyppijoukot eroavat toisistaan, joten viestityyppitunnisteiden avulla voidaan saada kokonaiskuva järjestelmän tilasta. Tutkimuksessa laskettiin jokainen eri viestityyppi, joka esiintyi järjestelmässä jokaisen 10 minuutin aikaikkunan sisällä. Lopputuloksena oli yli 400 viestityyppiä, joten tilasuhdevektorillakin on siis yli 400 ulottuvuutta.

Seuraavaksi muodostettuihin vektoreihin sovellettiin osiossa 4.3 esiteltyä menetelmää ja laskettiin neliöllistä ennustevirhettä jokaiselle aikaikkunalle. Tulokset näkyvät kuvassa 13. Kuvasta huomataan, että odotetusti taustatehtävien suhteellinen osuus pysyi vakiona aikaikkunoiden välillä, sillä tilasuhdevektoreissa ei näy poikkeavuuksia suurimmassa osassa aikaikkunoita. Kuvassa esiintyvät punaiset pallot piikeissä merkitsevät anomalioita. Nämä varmistettiin oikeiksi tapauksiksi GX:n monitorointijärjestelmästä, joka hälyttää, kun kuorma kasvaa tavallista suuremmaksi ja jonne merkitään kommentteja virheestä. Kuvassa näkyvät kommentit ovat näitä kommentteja. Koska artikkelissa esitetty menetelmä tulostaa epänormaalit lokihahmot, menetelmä voi auttaa suorituskykyongelmien diagnosoinnissa.

Viestimäärävektoreita voidaan käyttää yksittäisten taustatehtävien ja mahdollisesti järjestelmävirheiden havaitsemiseen. GX:ssä varasto on ositettu usealle solmulle. Osio luodaan, jaetaan solmuille ja lopuksi poistetaan uudelleenositus- tai poistooperaation seurauksena. Kun osiota muutetaan, se luo lokeihin lokiviestisarjoja. Jokaisella osiolla on tunniste, ja ne ryhmitellään kyseisen tunnisteen mukaan. Tunnisteen avulla voidaan tunnistaa lokiviestisarjat, ja yksittäinen lokiviestisarja voi



Kuva 13: Neliöllinen ennustevirhe kuvattuna (Xu ja muut, 2010).

yltää jopa koko tarkastellun kahden kuukauden ajalle. Tämän vuoksi viestisarjat jaotellaan vielä sessioittain kuten aiemmin esitellyssä tutkimuksessa (Xu ja muut, 2009b). GX:n tapauksessa vain alle 0,1% merkittiin anomalioiksi yli 200 miljoonan viestisarjan joukosta. Koska tutkimuksen tekijöillä ei ollut syvää tuntemusta GX:n logiikasta, ei rivejä voitu verifioida anomalioiksi, mutta he totesivat anomalioiksi merkittyjä viestisarjona tutkimalla, että pääkomponenttianalyysin esiin nostamat rivit olivat joko harvinaista viestityyppiä tai indikaatio sessiosta, jonka suoritus kesti tavanomaista pidempään.

5.4 Invariantit

HDFS:n lokidataa käytettiin myös invarianttimenetelmän suoriutumisen mittaamiseksi. Tutkimus suoritettiin testiympäristössä, jossa oli 15 solmua ja yhteensä 24 miljoonaa lokiviestiä. Palvelimilla ajettiin useita eri tyyppisiä esimerkkisovelluksia, kuten edellä mainittua WordCountia. Tässä tapauksessa testiympäristöön ei injektoitu virheitä, vaan katsottiin, mitä menetelmä löytää, ja varmistettiin tulosten oikeellisuus manuaalisesti (Lou ja muut, 2010).

Kuvaus	**Sarjojen määrä**
Tehtäviä epäonnistui, koska pulssi (heartbeat) katosi.	779
Tapettu tehtävä oli RUNNING tilassa JobTrackerilla ja TaskTrackerilla ikuisesti.	1 133
Useampaa kuin yhtä solmua pyydettiin replikoimaan datalohko yhdelle solmulle samanaikaisesti.	26
Datalohkon kirjoitus, vaikka se on jo olemassa.	25
Tehtävä JVM jumittui.	204
Vaihdettiin JVM, mutta merkittiin se tuntemattomaksi.	87
Vaihdettiin JVM ja poistettiin se välittömästi.	211
Yritettiin poistaa datalohko kun se oli auki asiakassovelluksella.	6
JVM epäjohdonmukainen tila.	416
PollForTaskWithClosed-Job kutsu JobTrackerilta TaskTrackerille aikakatkaistaan, kun tehtävä suoritetaan loppuun.	3

Taulukko 9: Löytyneiden anomalioiden kuvaukset ja tieto siitä, kuinka moni anomalian lokirivi oli kuvauksen mukainen (Lou ja muut, 2010).

Lokipohjien jäsentäminen tehtiin invarianttimenetelmän tapauksessa lokitiedostoista. Tämän jälkeen invarienttien etsiminen ja anomalioiden etsiminen tehtiin osiossa 4.4 mukaisesti ensin ryhmittelemällä viestit ja sitten etsimällä merkitykselliset invariantit invarianttiavaruudesta. Hadoopista tehdyssä tutkimuksessa löydettiin 36 invarianttia, joiden oikeellisuus varmistettiin käsin. Esimerkki löydetystä invariantista on $c(L_{113}) + c(L_{114}) = c(L_{90})$, jossa L_{113} on lokiviestityyppi "Choosing data-local task ##", L_{114}="Choosing rack-local task ##"ja L_{90} ="Adding task ## to tip ##, for tracker ##". Löydettyjen invarianttien joukossa ei ollut yhtään väärää positiivista invarianttia.

Menetelmä tunnisti 2 890 poikkeavaa lokisarjaa ja yhteensä 10 erilaista anomaliatyyppiä, jotka ovat listattu taulukossa 9. Mukana oli myös helposti huomaamatta jääviä virheitä kuten se, että Hadoop lähettää usealle solmulle pyynnön lähettää sama datalohko yhdelle solmulle replikoitavaksi. Solmu siis vastaanottaa enemmän lohkoja kuin se poistaa. Tämä rikkoo invarianttia count("Receiving block ##") = c("Deleting block file ##"), joten se tunnistetaan anomaliaksi. Virheellisiä positiivisia tunnistettiin 2 555, mutta uniikkeja virhetyyppejä oli yhteensä vain 2 kappaletta. Ensimmäinen vääristä positiivisista johtui Hadoopin aikataulutusominaisuudesta, joka käynnistää useita instansseja samasta tehtävästä. Näistä kuitenkin vain yksi suoritetaan loppuun asti ja muut ajot tapetaan välittämättä siitä, missä vaiheessa ne olivat. Toinen väärä positiivinen johtui töiden siivous- ja luontitehtävistä. Niiden lokiviestit muistuttavat map-tehtävän lokiviestejä, mutta eroaa kuitenkin tarpeeksi, että ne tulkitaan anomalioiksi.

6 Menetelmien vertailua

Jäsentämis- ja koneoppimismentelmiä on hyvä arvioida erillisinä kokonaisuuksina, sillä vaikka ne ovat esitellyissä artikkeleissa esitelty yleensä yhdessä, voi niitä käyttää hyvin myös erikseen. Menetelmien vertailua voidaan tehdä perustuen menetelmiä esitelleisiin tutkimuksiin, mutta myös olemassa oleviin vertailuihin. Menetelmiä vertaillaan niiden yleistettävyyden, suorituskyvyn ja suoriutumisen perusteella.

6.1 Jäsentämismenetelmien vertailua

Koneoppimismenetelmien suoriutumisen kannalta on tärkeää, että lokiviestit saadaan jäsennettyä tehokkaasti ja tarkasti. Artikkelissa (P. He ja muut, 2016) on vertailtu kahta osiossa 3.3 esiteltyä menetelmää SLCT ja LKE. Niiden suoriutumista tutkittiin viidessä eri järjestelmässä vaihtelevilla datamäärillä. Järjestelmät ja niiden lokiviestien määrä on listattu taulukossa 10. Mukana tutkimuksessa on kolmen tyyppisten järjestelmien lokidataa: HDFS ja Zookeeper ovat hajautettuja järjestelmiä, BGL ja HPC ovat supertietokoneita ja Proxifier itsenäinen ohjelmisto. Jäsentämismenetelmien tarkkuutta mitataan F-mitalla. Tätä varten lokitiedostot käydään läpi manuaalisesti, ja kun lokiviestimalli on saatu vahvistettua, sitä vastaavat viestit voidaan suodattaa käyttäen säännöllisiä lausekkeita. Jäsentämistarkkuuden mittaamiseksi lokiviesteistä valitaan satunnaisesti kaksi tuhatta jokaisesta järjestelmästä ja tarkkuuden mittana käytetään F-mittaa. Ennen menetelmien suoriutumisen tutkimista järjestelmien lokiviestejä on esiprosessoitu siten, että niistä on poistettu erilaisia tunnisteita järjestelmien tuntemuksen perusteella. Poistetut tunnisteet ovat esimerkiksi IP-osoitteita ja kuten osiossa 3.3 on esitetty, LKE-menetelmään se kuului alunperinkin.

Nimi	**Kuvaus**	**Lokirivien määrä**
HDFS	Hadoop hajautettu tiedostojärjestelmä	11 175 629
ZooKeeper	Hajautettu järjestelmäkoordinoija	74 380
BGL	BlueGene/L supertietokone	4 747 963
HPC	Korkean suorityskyvyn tietokoneryväs	433 490
Proxifier	Välityspalvelinasiakassovellus	10 108

Taulukko 10: Kuvaus tutkimuksessa käytetyistä järjestelmistä (P. He ja muut, 2016).

Molemmat jäsentämismenetelmät suoriutuvat melko hyvin, mutta SLCT suoriutui johdonmukaisesti hieman paremmin kuin LKE. Tarkemmat luvut löytyvät taulukosta 11. Taulukosta huomataan, että LKE suoriutui vain HDFS:n tapauksessa paremmin kuin SLCT. Sen tarkkuus oli myös huomattavasti huonompi kuin SLCT:n HPC:n lokitiedostojen jäsentämisessä. Tutkijoiden mukaan huono jäsentämistarkkuus HPC:n lokeissa johtui LKE:n aggressiivisesta ryvästämisestä eli siitä, että yhdistämiseen riittää, että ryvästen minkä tahansa kahden viestin etäisyys on pienempi kuin määritetty kynnysarvo. Myös suorituskyvyn suhteen SLCT vaikuttaa lupaavammalta: sillä meni 10 miljoonan HDFS:n lokiviestin jäsentämiseen vain viisi

minuuttia, kun taas LKE:n menetelmää ei edes voitu kokeilla kyseisellä datajoukolla sen hitauden vuoksi. Tutkimuksessa ei tosin käytetty hajauttamista, joka saattaisi nopeuttaa menetelmän suoritusta.

	HDFS	**Zookeeper**	**BGL**	**HPC**	**Proxifier**
SLCT	0,93	0,92	0,94	0,86	0,89
LKE	0,96	0,82	0,70	0,17	0,81

Taulukko 11: SLCT:n ja LKE:n suoriutumisen F-mitta eri järjestelmissä (P. He ja muut, 2016).

Jäsentämismenetelmän, jossa lokiviestipohjat jäsennetään lähdekoodista, suoriutumista ei ole tutkittu samassa tutkimuksessa kuin SLCT:n ja LKE:n, mutta menetelmän esitelleessä artikkelissa (Xu ja muut, 2009a) tehdyn tutkimukseen mukaan kyseisen menetelmän tarkkuus on 99,8%. Tämä on helppo uskoa, koska kyseinen menetelmä jäsentää lokiviestimallit lähdekoodista ja jokainen lokiviesti on peräisin jostain tulostuslausekkeesta. Menetelmän tarkkuutta nykyaikaisissa järjestelmissä saattaa huonontaa hieman ulkoisten kirjastojen käyttö, jolloin kaikki lokiviestien tulostuslausekkeet eivät löydykään varsinaisen järjestelmän lähdekoodista, vaan ne saattavat löytyä kirjastojen lähdekoodeista, joita ei tässä tapauksessa jäsennetä. Menetelmä on ratkaissut ongelman siten, että että se etsii yleisimpien projektien lähdekoodeista vastaavuutta, jos viestipohjaa ei löydy alkuperäisen järjestelmän lähdekoodista muodostetuista viestipohjista. Suorituskyky on selkeästi järjestelmän ohjelmointikielestä riippuvaista. Kielissä, joissa tulostuslausekkeista on suoraan nähtävissä, onko kyseessä lokituslauseke ja mitä lokituslauseke tulostaa, on suoraviivaista jäsentää lokiviestien mallit. Olio-ohjelmointikielien tapauksessa saattaa taas joutua käymään läpi joukkohierarkioita muodostaakseen lopullisen lokiviestimallin. HDFS on kuitenkin Java-pohjainen järjestelmä, ja artikkelissa (Xu ja muut, 2009a) kerrotaan, että 50 solmulla jäsentämisessä meni alle 3 minuuttia ja kymmenelläkin alle kymmenen minuuttia, joten sen suoriutuminen oli hyvä olio-ohjelmointikielestä huolimatta. Hajautettuna menetelmä siis suoriutuu jäsentämisestä todella nopeasti, eikä sen suoritus todennäköisesti vaikuta ratkaisevasti kokonaismenetelmän suorityskykyyn. Toisaalta jos menetelmää ei hajauta, menetelmän suorituskyky saattaa olla huonompi. Artikkelissa (Xu ja muut, 2009a) mainittiin, että lähdekoodissa esiintyvistä lokituslausekkeista syntyvistä lokiviesteistä vain 1-5% esiintyy tuotannon lokitiedostoissa, koska suurin osa lokituslausekkeista suoritetaan vain virheiden sattuessa. Lisäksi kehitys- ja tuotantoympäristöissä käytetään erilaisia konfiguraatioita, joiden avulla voidaan hallita, minkä tasoisia tapahtumia lokitetaan. Osittain siis saatetaan tehdä hieman turhaa työtä, kun koko lähdekoodi käydään läpi.

Jokaisella edellä esitellyllä menetelmällä on etunsa, ja mahdollisuus käyttää niitä on pitkälti olosuhteista riippuvainen. Esimerkiksi lähdekoodin käyttäminen lokiviestien mallin lähteenä edellyttää luonnollisesti lähdekoodin saatavuutta. Viestimallien louhiminen lokitiedostoista taas näkee sovelluksen mustana laatikkona, joka ei vaadi kun että olemassa olevia lokitiedostoja on saatavilla. Sekä SCLT että lähdekoodista jäsentäminen vaikuttavat molemmat tehokkailta ja tarkoilta tavalta jäsentää loki-

viestien mallit, mutta lähdekoodista jäsentäminen ei ole samalla tavalla yleistettävä kuin SLCT. Esimerkiksi konsulttiyrityksessä tehdään useita järjestelmiä, joten jokaiselle eri ohjelmointikielellä tehdylle järjestelmälle jouduttaisiin tehdä oma työkalunsa. Tämä ei vaikuta kovinkaan tehokkaalta tavalta. Lisäksi jo aiemmin mainittu ulkoisten kirjastojen käyttö hankaloittaa jäsentämistä. Lähdekoodiin perustuvaa menetelmää voisi varmaankin siis käyttää jokin tuotetalo, jolla tuote pohjautuu yhteen ohjelmointikieleen ja joka ei käytä juurikaan ulkoisia kirjastoja.

Lokitiedostoista suoritettava jäsentäminen vaikuttaa paremmalta vaihtoehdolta, erityisesti SLCT. Verrattuna LKE:hen sen suorituskyky on todella hyvä, koska se kasvaa lineaarisesti (P. He ja muut, 2016). Niitä vertailevassa tutkimuksessa ei tosin ollut hajautettu menetelmien suoritusta, joten LKE:n suorituskyky näyttää ehkä huonommalta kuin se olisi hajautettuna. K-keskiarvot (K-means) ryvästämisen hajauttaminen on mahdollista, ja se on implementoitu esimerkiksi Apache Sparkissa (Apache Software Foundation, 2018). K-keskiarvot vaikuttaa LKE:ssä suoritusaikaan huomattavasti, joten sen hajauttaminen saattaisi nopeuttaa menetelmän suoritusta. Toisaalta SLCT:n suoriutuminen on myös yleensä parempi kuin LKE:n eikä vaihtele yhtä paljon. Tässäkin mielessä SLCT voisi siis olla näistä kahdesta menetelmästä parempi valinta, koska valitun menetelmän olisi hyvä toimia erilaisissa järjestelmissä. SLCT:n tarkkuus pysyy kaikkien tutkimuksen (P. He ja muut, 2016) järjestelmien osalta yli 80%:n, kun lokitiedostoja on esiprosessoitu.

Jos lokiviestit tuotetaan ulkoisella ohjelmistolla, joka tuottaa strukturoituja ja tiedossa olevia lokiviestejä, lokiviestien pohjien jäsentäminen ei ole edes tarpeen. Väliohjelmistojen käyttö ei riipu ohjelmointikielestä silloin, kun koodia injektoidaan binäärikoodiin tai kun ohjelmisto toimii täysin erillisenä komponenttina, mutta vaatii pääsyn palvelimelle, jossa sovellus on käynnissä. Täysin irrallisena komponenttina se voi tietysti tuottaa vain rajatun tyyppisiä lokitietoja, esimerkiksi ohjelmiston lähettämiä tai vastaanottamia pyyntöjä. Ulkoinen ohjelmisto saattaa myös heikentää suorituskykyä järjestelmässä, johon se on injektoitu. Jos lokitus lisätään järjestelmän lähdekoodiin, on tärkeää, että tärkeät kohdat on lokitettu, jotta lokeja voidaan käyttää ongelmien selvittämiseen.

6.2 Koneoppimismenetelmien vertailua

Esiteltyjen koneoppimismenetelmien suoriutumista on vertailtu tutkimuksessa (S. He ja muut, 2016). Tutkimus vertailee osiossa 4.1 esitellyn LogClusterin, osiossa 4.3 esitellyn pääkomponenttianalyysin (Xu ja muut, 2009a) ja osiossa 4.4 esitellyn invarianttimenetelmän suoriutumista kahdesta eri järjestelmästä kerätyllä datajoukolla. Mukana datajoukossa on yhteensä 15 923 592 lokiviestiä ja 365 298 anomaliainstanssia. Tutkimuksessa mitataan eri menetelmien tarkkuutta eri metriikoilla ja lisäksi suorituskykyä.

Toinen järjestelmistä on HDFS ja toinen BGL eli BlueGene/L supertietokonejärjestelmä. Kaikki data on kerätty tuotannosta ja jokaiselle riville on tehty luokittelu manuaalisesti alkuperäisten asiantuntijoiden toimesta. Näitä luokittelutietoja käyte-

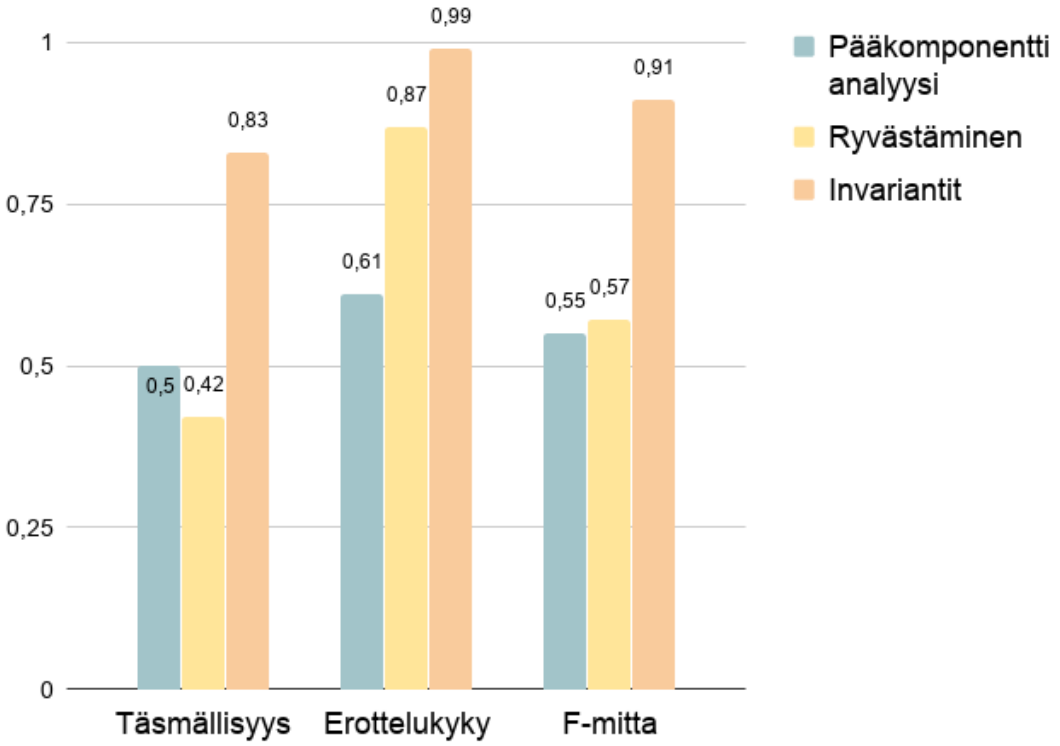
tään vain tulosten mittaamiseen. HDFS:stä on kerätty dataa seitsemän kuukauden ajalta ja se sisältää 11 175 629 lokiviestiä ja on kooltaan 1,55 gigatavua. Lokiviestit kuvastavat tiedostojärjestelmässä suoritettuja lohko-operaatioita kuten muistin varaaminen, kirjoitus, replikointi tai poisto. Jokainen näistä viesteistä sisältää myös uniikin lohkotunnisteen. BGL:stä kerätty data ei sitä vastoin sisällä tunnisteita jokaiselle operaatioille. Se sisältää 4 747 963 lokiviestiä, joista 348 460 lokiviestiä on merkitty anomaliaksi. Datan koko on 807 megatavua ja se on kerätty 38,7 tunnin aikana.

Menetelmien tarkkuutta mitataan täsmällisyydellä, erottelukyvyllä ja F-mitalla. HDFS:n kohdalla käytetään sessioikkuna-menetelmää ja BGL:n kohdalla liukuva ikkuna -menetelmää. Ryvästäminen on erittäin hidas menetelmä HDFS:n suurella datamäärällä, joten kaikkea dataa ei voida käyttää sen tutkimiseen. Kaikkien menetelmien täsmällisyys on hyvä HDFS:n kohdalla, mutta BGL:n kohdalla vain invariantteihin perustuvan menetelmän täsmällisyys on hyvä. Erot BGL:n datajoukolla näkyvät diagrammissa 14. Artikkelissa on tutkittu myös päätöspuita, logistista regressiota ja tukivektorikoneita eli ohjatun koneoppimisen menetelmiä. Niihin verrattuna mikään ohjaamattoman koneoppimisen menetelmä ei pärjää hyvin, mutta invariantit suoriutuvat kuitenkin hyvin sekä HDFS:n että BGL:n datajoukoilla. Tarkemmat tulokset invarianttimenetelmälle näkyvät taulukossa 12. Artikkelin kirjoittajien mukaan ryvästämisen huonompi tarkkuus johtuu todennäköisesti siitä, että tapahtumamäärämatriisi on harva ja siinä on suuri määrä ulottuvuuksia. Pääkomponenttianalyysin huono suoriutuminen BGL:n datalla johtuu tutkijoiden selvityksen mukaan siitä, että anomalioiden ja normaalien tapahtumien erottamista ei voida suorittaa yhden kynnysarvon mukaan.

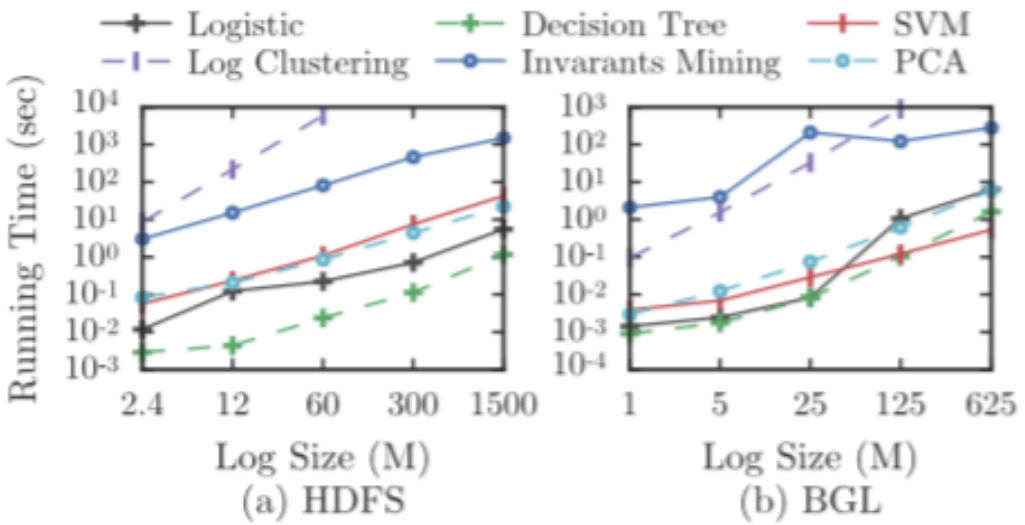
Täsmällisyys	HDFS	88%
	BGL	83%
Erottelukyky	HDFS	95%
	BGL	99%
F-mitta	HDFS	91%
	BGL	91%

Taulukko 12: Invariantteihin perustuvan menetelmän suoriutuminen (S. He ja muut, 2016).

Myös menetelmien suorituskykyä tutkittiin. Tutkimuksessa vaihdeltiin käytettävän datan määrää eri menetelmillä ja huomattiin, että ohjaamattoman koneoppimisen menetelmät pääkomponenttianalyysiä lukuunottamatta ovat huomattavasti tehottomampia kuin ohjatun koneoppimisen menetelmät. Kuvassa 15 näkyy eri menetelmien suoritusajat eri datamäärillä. Tutkimus suoritettiin Linux-palvelimella, jossa oli Intel Xeon E5-2670v2 CPU ja 128GB DDR3 1600 RAM ja käyttöjärjestelmänä Ubuntu 14.04.2 Linux Kernel 3.16.0 versiolla. Artikkelissa ei ole mainittu, että menetelmien suoriutumista olisi mitattu hajauttamalla menetelmiä, eikä tutkijoiden tekemän työkalun lähdekoodikaan anna viitteitä siitä (Logpai, 2016).



Kuva 14: Eri menetelmien tarkkuus BGL:n datajoukolla (S. He ja muut, 2016).



Kuva 15: Suorituskykytuloksia eri menetelmillä (S. He ja muut, 2016).

Pääkomponenttianalyysin ja invarianttimenetelmän suoritusaika kasvaa lineaarisesti, kun taas ryvästämisen aikavaativuus on $O(n^2)$ (S. He ja muut, 2016). Kuten jo aiemmin todettiin, ryvästämisen vaatima suoritusaika kasvaa liian suureksi HDFS:n tarjoamilla datamäärillä rajoittaen sen potentiaalia. Invarianttien suoritusaika on BGL:n datalla suurempi kuin ryvästämismenetelmällä koska BGL:n datassa on enemmän tapahtumajoukkoja kuin HDFS:n datassa. Koska menetelmässä on määritelty lopetusehto, jonka avulla voidaan kontrolloida raaka voima -etsintäprosessia suurilla datajoukoilla, kasvu pysähtyy kun saavutetaan 125 megatavun datan koko.

Painotusmenetelmästä suoritetussa tutkimuksessa ei käsitellä sen suorituskykyä, mutta sen voidaan päätellä olevan hyvä, koska matriisin muodostamisen jälkeen data käydään vain kerran läpi ja lasketaan painotukset. Osion 5.2 perusteella voidaan päätellä, että menetelmä ei suoriudu kovinkaan hyvin. Menetelmän suoriutumista on mitattu täsmällisyydellä, erottelukyvyllä ja F-mitalla. Vaikka tutkijat artikkeleissa (Stearley & Oliner, 2008) ja (Oliner ja muut, 2008) vaikuttavat tyytyväisiltä tuloksiinsa ja kertovat Nodeinfon olevan tuotantokäytössä, tulokset verrattuna muihin menetelmiin ovat huonot. F-mitta on parhaimmillaankin vain $\frac{2\times0,75\times0,5}{0,75+0,5} = 0,6$ (Stearley & Oliner, 2008) ja Nodeinfoa käsittelevässä tutkimuksessa vain noin 0,54. Lisäksi he käyttävät vertailukohteena todella yksinkertaista, itse määrittelemäänsä menetelmää, joka havaitsee vain 10% anomalioista (Stearley & Oliner, 2008). Tämän ylittäminen ei ole vielä osoitus siitä, että menetelmä olisi hyvä. Toki menetelmiä kannattaisi verrata aina niin, että menetelmiä käytetään samassa järjestelmässä samalla datalla, mutta koska se ei ole tässä tapauksessa mahdollista, tehdään vertailu perustuen alkuperäisen tutkimuksen mittoihin.

Myöskään LogCluster ei vaikuta lupaavalta sen esittelevän artikkelin (Lin ja muut, 2016) tai vertailevan tutkimuksen (S. He ja muut, 2016) perusteella. Alkuperäisessä tutkimuksessa menetelmän keskitäsmällisyyden kerrotaan olevan noin 60% ja Microsoftin tuotantojärjestelmissä se on jopa sitäkin huonompi, Palvelu X:ssä 42,86% ja Palvelu Y:ssä 55%. Vertailevassa tutkimuksessa täsmällisyys oli 0,42%, erottelukyky 0,87% ja F-mitta 0,57% BGL:n datalla, joten menetelmä ei pärjää esimerkiksi invarianteille. Täsmällisyyden lisäksi LogClusterin suorituskyky ei ole sillä tasolla, että sitä voisi hyödyntää suurilla datamäärillä. Vertailevan tutkimuksen mukaan sen aikavaativuus on $O(n^2)$ ja HDFS:n datalla sen suoritus kasvaa jo yli kahteen tuntiin 60 megatavun määrällä. Toisen ryvästämismenetelmän, Pinpointin, suoriutumista ei ole tutkittu vertailevasti, mutta sen esittelevässä artikkelissa (Chen ja muut, 2002) tulokset ovat kohtuullisia. Tarkkuus on jopa 80-90%, mutta toisaalta täsmällisyys on 50-60%. Sekään ei siis pärjää invarianttimenetelmälle. Toinen ongelma menetelmässä on se, että se on kokonainen työkalu lokien keräyksestä niiden analysointiin, joten analysointia ei voi käyttää ilman sovellukseen lisättävää lokitusta. Muita ulkoisen ohjelmiston käyttöön liittyviä rajotuksia on käsitelty aiemmin.

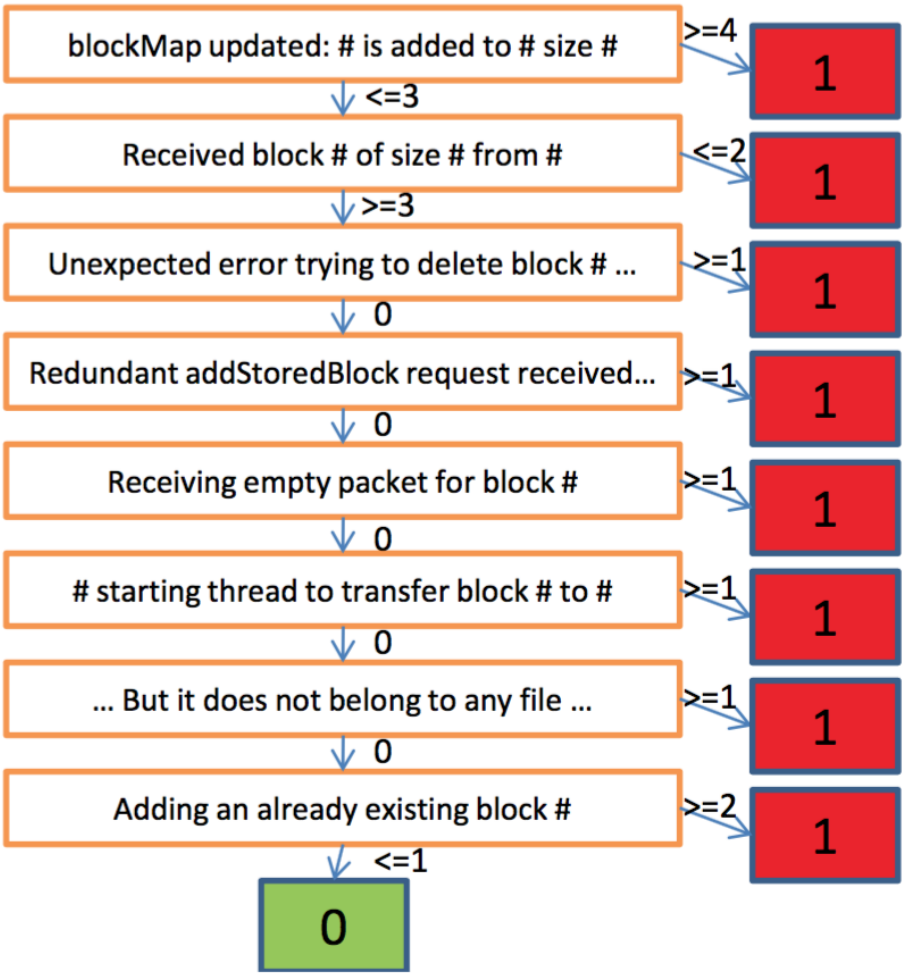
Epäilypisteisiin perustuvan ryvästysmenetelmän suoriutumista ei ole järjestelmällisesti tutkittu, ja menetelmän esitelleessä artikkelissa (Mirgorodskiy ja muut, 2006) todetaan vain, että menetelmä oli tuotantokäytössä ja että sen avulla oli onnistuttu tunnistamaan joitakin anomalioita. Koska menetelmän suoriutumisesta ei ole lisätietoja artikkelissa, ei voida tietää kuinka paljon se on löytänyt anomalioita ja

kuinka paljon se on antanut vääriä positiivisia. Artikkeli ei sisällä myöskään pohdintaa menetelmässä esiintyneistä ongelmista ja kehityskohteista, joten on vaikeaa arvioida, miten hyvin menetelmä todellisuudessa toimii. Menetelmään liittyy myös sama ongelma kuin Pinpointiin: siinä oletetaan käytettävän ulkopuolista toimijaa, joka tuottaa ennaltamäärätyn kaltaisia lokiviestejä. Verrattuna Pinpointiin etuna on kuitenkin se, että menetelmän toimija toimii täysin erillisenä ja muokkaa vain binäärikoodia, joten se ei ole riippuvainen ohjelmointikielestä. Tästä huolimatta mikään esitetyistä ryvästämismenetelmistä ei vaikuta kovin lupaavalta.

Invarianttimenetelmä vaikuttaa siis kokonaisuutena lupaavimmalta menetelmältä. Täysin ongelmaton sekään ei ole, sillä se ei ole yhtä tehokas kuin pääkomponenttianalyysi, mutta toisaalta se vaikuttaisi tutkimuksen (S. He ja muut, 2016) perusteella olevan yleistettävämpi kuin pääkomponenttianalyysi, sillä se suoriutuu paremmin molemmissa tutkimuksessa käytetyissä järjestelmissä. Sekä invarianttimenetelmä että pääkomponenttianalyysi muodostavat viestimäärävektoreita ja pyrkivät niiden perusteella muodostamaan suorituspolkuja, joten perusidea on niissä sama. Pääkomponenttianalyysi ei kuitenkaan tulkitse viestityyppien välisiä suhteita, joten se ei havaitse kaikkia anomalioita, jotka invarianttimenetelmä havaitsee (S. He ja muut, 2016). Pääkomponenttianalyysin ja invarianttimenetelmän välillä on myös eroa siinä, miten anomalioita voidaan tulkita. Pääkomponenttianalyysin avulla löydettyjen anomalioiden ymmärtämiseksi pitää käydä lähdekoodia läpi manuaalisesti. Tutkijat Fu ja muut ehdottivat, että menetelmän tuloksista voitaisiin muodostaa päätöspuita, ja yksi esimerkki muodostettavasta päätöspuusta on kuvassa 16. Kuvassa nuolien vieressä esiintyvät määrät ovat kynnysarvoja viestimäärille, ja punaisella taustalla oleva numero yksi tarkoittaa, että ehdon perusteella todetaan poikkeavuus. Siis esimerkiksi jos lokiviestiä 'Received block # of size # from #' on vähemmän kuin tai tasan kaksi kappaletta, kyseessä on anomalia. Invarianttimenetelmä taas palauttaa suoraan rikotun invariantin, jonka avulla voidaan ymmärtää poikkeavuus. Jos rikottu invariantti on c('Replicated block # to node #')=c('Deleted block #'), on helppo tulkita, että joko replikoimisessa tai lohkon poistamisessa on ongelma.

Toisaalta invarianttimenetelmän kohdalla ei ole otettu kantaa siihen, miten menetelmä toimisi niin, että anomaliat voitaisiin havaita liki reaaliajassa (Lou ja muut, 2010). Invarianttimenetelmään olisi kuitenkin todennäköisesti sovellettavissa pääkomponenttianalyysin ratkaisu tähän asiaan. Tätä varten voitaisiin määritellä kaksi aikarajaa T_{max} ja $T_{katkaisu}$, jotka määrittelisivät, kuinka kauan suorituspolun muodostumista odotetaan. Kuten aiemmin luvussa 3 mainitaan, $T_{katkaisu}$ lasketaan suoritusajan jakaumasta, jonka voisi invarianttien tapauksessa muodostaa esimerkiksi invariantteja vastaavien viestiryhmien ensimmäisen ja viimeisen tapahtuman aikaleimojen erotuksena. Sen lisäksi, että tämä auttaisi nopeassa anomalioiden havaitsemisessa, invarianttimenetelmä voisi myös tämän avulla havaita suorituskykyongelmia kuten pääkomponenttianalyysi tekee.

Invarianttimenetelmän suorituskykyä ja siihen liittyviä haasteita on pohdittu menetelmää esittelevässä artikkelissa (Lou ja muut, 2010). Siinä todetaan, että harvojen invarianttivektorien löytäminen on yleisesti ottaen NP-kova ongelma. Osiossa 4.4 esitetyn menetelmän aikavaativuus on $O(\sum_{i=1}^{m-r+1} C_m^i)$, jossa C_m^i merkitsee kompaktia



Kuva 16: Pääkomponenttianalyysin tulokset visualisoituna (Xu ja muut, 2009a).

invarianttijoukkoa, jossa on i nollasta poikkeavaa kerrointa ja m on kerrointen määrä. Kyseinen aikavaativuus ei ole kovin pieni, jos matriisin $\mathbf{X}$ riviavaruus $m-r+1$ on suuri. Tutkijat kertovat kuitenkin artikkelissa, että sovellusten riviavaruuksien ulottuvuus on yleensä hyvin pieni, jolloin suoritusaika ei kasva liian suureksi. Useiden sovellusten harvat invariantit sisältävät myös vain 2-3 nollasta poikkeavaa kerrointa. Koska riippumattomia invariantteja on enintään r kappaletta, yli 5 nollasta poikkeavan kertoimen kombinaatioiden etsintää ei tarvitse suorittaa jos on jo löydetty r riippumatonta invarianttia, kun $k < 5$.

Laskennallista tehokkuutta voidaan parantaa myös ohittamalla joidenkin hypoteesikandidaattien etsiminen. Koska kaikki invarianttien lineaariset kombinaatiot ovat myös invariantteja, jo löydettyjen invarianttien lineaarisia kombinaatioita ei tarvitse enää etsiä. Myös singulaariarvohajotelman korvaaminen ominaisarvohajotelmalla auttaa skaalaamaan järjestelmää oikeanpuoleisten singulaarivektoreiden laskemiseen, koska singulaariarvohajotelma on ominaisarvohajotelman yleistys. Invarianttikandidaattien θ löytäminen voidaan siis myös tehdä ominaisarvohajotelmalla ja ominaisarvohajotelma voidaan myös hajauttaa. Tukisuhdeluvun laskeminen voidaan myös hajauttaa helposti ja näin parantaa suoritusaikaa (Lou ja muut, 2010). Tämä voitaisiin tehdä mapReduce-tyyppisesti niin, että map-vaiheessa lasketaan invarianttien esiintyminen yhdessä lokiviestiryhmässä ja reduce-vaiheessa nämä yhdistettäisiin.

Koska mallit on ohjaamattoman hahmontunnistuksen avulla muodostettu, harvinaiset tapahtumat mielletään menetelmissä anomalioiksi. Tästä syystä mallin päivityksen tiheys on erityisen tärkeä kysymys etenkin tulevaisuudessa, mutta jo nytkin. Järjestelmistä saatetaan julkaista uusia versioita uusilla ominaisuuksilla päivittäin, ja kun esimerkiksi invarianttimenetelmä löytää anomaliat muodostamalla suorituspolkuja, uudet tapahtumat merkitään todennäköisesti anomaliaksi. Tästä syystä mallien päivitys tulee sovittaa julkaisurytmiin. Harvinaisten tapahtumien merkitseminen anomaliaksi tarkoittaa myös sitä, että harvinaiset, mutta normaalit tapahtumat merkitään anomalioiksi tarpeettomasti. Tämän vuoksi olisi myös hyvä olla mahdollisuus merkitä anomalioita vääriksi positiivisiksi, jolloin samaa harvinaista tapahtumaa ei merkittäisi anomaliaksi useampaa kertaa.

7 Yhteenveto

Anomaliat lokitiedostoissa voivat paljastaa suorituksen aikaisia ongelmia, jotka johtuvat esimerkiksi toimintaympäristöstä tai virheistä ohjelmakoodissa. Lokitiedostojen tehokas hyödyntäminen onkin tärkeää ja mitä suuremmasta järjestelmästä on kyse, sitä tärkeämpää se on. Koska lokitiedostoissa ei ole yleensä tietoa siitä, sisältääkö jokin tapahtuma tai tapahtumasarja anomalioita, on ohjaamaton koneoppiminen luontevin tapa havaita anomaliat.

Suoritusanomalioiden havaitseminen lokitiedostosta vaatii ensin lokiviestien esiprosessointia, koska viestit ovat epästrukturoitua dataa. Yleensä lokiviesteistä muodos-

tetaan lokiviestipohjat, joissa viesteissä esiintyvät muuttujat on poistettu ja jäljellä on vain vakio-osa. Tämä voidaan tehdä olemassaolevien lokitiedostojen tai lähdekoodin perusteella. Lokiviestipohjien jäsentämisen jälkeen muodostetaan lokisarjat ja niistä edelleen numeeriset matriisit. Lokiviestipohjien muodostavia menetelmiä ovat lokiviestipohjien jäsentäminen lähdekoodista ja kaksi lokitiedostoista pohjat jäsentävää menetelmää LKE ja SLCT. Lokien esiprosessointimenetelmistä lupaavimmaksi osoittautui tehokas ja tarkka SLCT (Vaarandi, 2003), joka jäsentää lokiviestipohjat lokitiedostoista ryvästämällä.

Tutkielmassa esitellyt koneoppimismenetelmät ovat ryvästäminen, painotusmenetelmät, pääkomponenttianalyysi ja invariantit. Menetelmiä on arvioitu niiden yleistettävyyden, suorituskyvyn ja tarkkuuden perusteella. Ryvästäminen ja painotusmenetelmät eivät menestyneet vertailussa niiden huonon tarkkuuden vuoksi, minkä lisäksi ryvästämisen todettiin olevan liian huono suorituskyvyltään. Pääkomponenttianalyysi on tehokas menetelmä, joka ylsi hyvään tarkkuuteen menetelmän esitelleiden artikkelien tutkimuksissa, mutta vertailevassa tutkimuksessa sen tarkkuuden havaittiin vaihtelevan eri järjestelmien välillä. Luvussa 5 esiteltyjen tutkimusten perusteella lupaavimmaksi menetelmäksi nousi invarianttimenetelmä, jossa järjestelmän suorituspolkuja kuvataan yhtälöiden avulla. Sen tarkkuus erilaisissa järjestelmissä oli johdonmukaisesti korkea ja suorituskykyäkin saa parannettua menetelmän esittelevässä artikkelissa (Lou ja muut, 2010) esitellyillä menetelmillä ja hajauttamalla. Optimoinnin lisäksi menetelmälle pitää määritellä, miten pitkä viive anomalioiden havaitsemiselle sallitaan ja miten anomalioiden havaitseminen tuotantokäytössä käytännössä toimii.

Konsulttiyritykselle toimivalta yhdistelmältä vaikuttaa siis ensin lokitiedostojen jäsentäminen käyttäen SLCT-menetelmää ja sitten tämän pohjalta invarianttimenetelmän käyttämisen anomalioiden havaitsemiseksi. Web-sovellukset ja analyytikkasovellukset ovat todennäköisesti yleisimpiä projektityyppeja tyypillisissä ohjelmistoalan konsulttiyrityksissä. Tutkimusten joukossa ei ollut lainkaan niiden tyyppisiä sovelluksia, joten menetelmien soveltuvuuden validoimiseksi pitäisi tutkia niiden suoriutumista vastaavissa järjestelmissä.

Viitteet

Apache Software Foundation. (2018). *Apache Spark, Clustering.* `https://spark.apache.org/docs/latest/mllib-clustering.html`. (Accessed: 2018-04-27)

Bonaccorso, G. (2017). Machine learning algorithms. Teoksessa (s. 181–183,208–211). Packt Publishing Ltd.

Chandola, V., Banerjee, A., & Kumar, V. (2009). Anomaly detection: A survey. *ACM computing surveys (CSUR)*, *41*(3), 15.

Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., & Brewer, E. (2002). Pinpoint: Problem determination in large, dynamic internet services. Teoksessa *International Conference on Dependable Systems and Networks, 2002. DSN 2002. Proceedings.* (s. 595–604).

Donoho, D. L., Tsaig, Y., Drori, I., & Starck, J.-L. (2012). Sparse solution of underdetermined systems of linear equations by stagewise orthogonal matching pursuit. *IEEE transactions on Information Theory*, *58*(2), 1094–1121.

Dunia, R., & Qin, S. J. (1997). Multi-dimensional fault diagnosis using a subspace approach. Teoksessa *American Control Conference.*

Fu, Q., Lou, J.-G., Wang, Y., & Li, J. (2009). Execution anomaly detection in distributed systems through unstructured log analysis. Teoksessa *Ninth IEEE International Conference on Data Mining, 2009. ICDM'09.* (s. 149–158).

Fu, Q., Zhu, J., Hu, W., Lou, J.-G., Ding, R., Lin, Q., ... Xie, T. (2014). Where do developers log? an empirical study on logging practices in industry. Teoksessa *Companion proceedings of the 36th international conference on software engineering* (s. 24–33).

He, P., Zhu, J., He, S., Li, J., & Lyu, M. R. (2016). An evaluation study on log parsing and its use in log mining. Teoksessa *46th annual ieee/ifip international conference on dependable systems and networks (dsn), 2016* (s. 654–661).

He, S., Zhu, J., He, P., & Lyu, M. R. (2016). Experience report: system log analysis for anomaly detection. Teoksessa *IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), 2016* (s. 207–218).

Lin, Q., Zhang, H., Lou, J.-G., Zhang, Y., & Chen, X. (2016). Log clustering based problem identification for online service systems. Teoksessa *Proceedings of the 38th international conference on software engineering companion* (s. 102–111).

Logpai. (2016). *Loglizer.* `https://github.com/logpai/loglizer`. (Accessed: 2018-05-09)

Lou, J.-G., Fu, Q., Yang, S., Xu, Y., & Li, J. (2010). Mining invariants from console logs for system problem detection. Teoksessa *USENIX Annual Technical Conference.*

Mariani, L., & Pastore, F. (2008). Automated identification of failure causes in system logs. Teoksessa *19th International Symposium on Software Reliability Engineering, 2008. ISSRE 2008.* (s. 117–126).

Mirgorodskiy, A. V., Maruyama, N., & Miller, B. P. (2006). Problem diagnosis in large-scale computing environments. Teoksessa *Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (s. 88).

Oliner, A. J., Aiken, A., & Stearley, J. (2008). Alert detection in system logs. Teoksessa *Eighth IEEE International Conference on Data Mining, 2008. ICDM'08.* (s. 959–964).

Robert, C. (2014). *Machine learning, a probabilistic perspective.* Taylor & Francis.

Shang, W., Jiang, Z. M., Hemmati, H., Adams, B., Hassan, A. E., & Martin, P. (2013). Assisting developers of big data analytics applications when deploying on hadoop clouds. Teoksessa *Proceedings of the 2013 International Conference on Software Engineering* (s. 402–411).

Stearley, J., & Oliner, A. J. (2008). Bad words: Finding faults in spirit's syslogs. Teoksessa *8th IEEE International Symposium on Cluster Computing and the Grid, 2008. CCGRID'08.* (s. 765–770).

Tibshirani, R., James, G., Witten, D., & Hastie, T. (2013). *An introduction to statistical learning-with applications in r.* New York, NY: Springer.

Vaarandi, R. (2003). A data clustering algorithm for mining patterns from event logs. Teoksessa *3rd ieee workshop on ip operations & management, 2003.(ipom 2003).* (s. 119–126).

Wold, S., Esbensen, K., & Geladi, P. (1987). Principal component analysis. *Chemometrics and intelligent laboratory systems*, *2*(1-3), 37–52.

Xu, W., Huang, L., Fox, A., Patterson, D., & Jordan, M. (2009b). Online system problem detection by mining patterns of console logs. Teoksessa *Ninth IEEE International Conference on Data Mining, 2009. ICDM'09.* (s. 588–597).

Xu, W., Huang, L., Fox, A., Patterson, D., & Jordan, M. I. (2009a). Detecting large-scale system problems by mining console logs. Teoksessa *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (s. 117–132).

Xu, W., Huang, L., & Jordan, M. I. (2010). Experience mining google's production console logs. Teoksessa *Proceedings of 2010 workshop on managing systems via log analysis and machine learning techniques.*